Query-Driven Indexing in Large-Scale Distributed Systems

THÈSE N^{O} 4280 (2009)

PRÉSENTÉE LE 30 JANVIER 2009

À LA FACULTE INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES D'INFORMATION RÉPARTIS
SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Gleb SKOBELTSYN

acceptée sur proposition du jury:

Prof. A. Ailamaki, présidente du jury
Prof. K. Aberer , directeur de thèse
Prof. R. Baeza-Yates, rapporteur
Prof. M. Henzinger, rapporteur
Dr F. Silverstri, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Lausanne, EPFL
2009

To my parents Elena and Kirill.

Abstract

Efficient and effective search in large-scale data repositories requires complex indexing solutions deployed on a large number of servers. Web search engines such as *Google* and *Yahoo!* already rely upon complex systems to be able to return relevant query results and keep processing times within the comfortable sub-second limit. Nevertheless, the exponential growth of the amount of content on the Web poses serious challenges with respect to scalability. Coping with these challenges requires novel indexing solutions that not only remain scalable but also preserve the search accuracy.

In this thesis we introduce and explore the concept of *query-driven indexing* – an index construction strategy that uses caching techniques to adapt to the querying patterns expressed by users. We suggest to abandon the strict difference between indexing and caching, and to build a distributed indexing structure, or a distributed cache, such that it is optimized for the current query load.

Our experimental and theoretical analysis shows that employing query-driven indexing is especially beneficial when the content is (geographically) distributed in a Peer-to-Peer network. In such a setting extensive bandwidth consumption has been identified as one of the major obstacles for efficient large-scale search. Our indexing mechanisms combat this problem by maintaining the query popularity statistics and by indexing (caching) intermediate query results that are requested frequently. We present several indexing strategies for processing multi-keyword and XPath queries over distributed collections of textual and XML documents respectively. Experimental evaluations show significant overall traffic reduction compared to the state-of-the-art approaches.

We also study possible query-driven optimizations for Web search engine architectures. Contrary to the Peer-to-Peer setting, Web search engines use centralized caching of query results to reduce the processing load on the main index. We analyze real search engine query logs and show that the changes in query traffic that such a results cache induces fundamentally affect indexing performance. In particular, we study its impact on index pruning efficiency. We show that combination of both techniques enables efficient reduction of the query processing costs and thus is practical to use in Web search engines.

Keywords: large-scale systems, query-driven indexing, P2P, P2PIR, information retrieval, XML, Web search engines, caching, index pruning.

Résumé

Des techniques d'indexation complexes, déployées sur un grand nombre de serveurs, sont nécessaires pour garantir l'efficacité des techniques de recherche d'information utilisées au sein de répertoires de données de grande taille. Par exemple, des moteurs de recherche comme *Google* ou *Yahoo!* s'appuient sur de telles techniques pour retourner des résultats de recherche pertinents tout en préservant des temps de traitement substanciellement inférieurs à la seconde. Toutefois, la croissance exponentielle de la quantité d'information présente sur le Web soulève des défis de passage à l'échelle et de précision qui necéssitent des solutions novatrices pour l'indexation.

L'objectif de ce travail de thèse est de présenter et d'analyser le concept d'indexation guidée par les requêtes – une stratégie d'indexation utilisant des techniques de cache afin de s'adapter aux modèles de requêtage exprimés par les utilisateurs. En particulier, nous proposons d'abandonner la distinction entre indexation et cache, et de construire une structure d'indexation distribuée sous la forme de caches distribués optimisant la charge de requêtage.

Notre analyse expérimentale et théorique montre que l'utilisation de telles techniques d'indexation guidées par les requêtes est particulièrement utile lorsque le contenu du répertoire est (géographiquement) distribué au sein d'un réseau pair-à-pair. En effet, dans ce type de situations, une consommation excessive de bande passante a été identifiée comme l'un des obstacles majeurs pour la mise en oeuvre efficace de systèmes de recherche d'information de grande taille. Les mécanismes d'indexation que nous proposons permettent d'apporter une solution à ce problème en maintenant les statistiques de popularité des requêtes et en indexant (mise en cache) les résultats intermédiaires des recherches fréquentes. Nous proposons plusieurs mécanismes d'indexation pour le traitement de requêtes multi-termes (resp. de requêtes XPath) opérant sur des collections distribuées de documents textuels (resp. de documents XML). Nos évaluations expérimentales démontrent une significative réduction du trafic global en comparaison des approches les plus récentes correspondant à l'état de l'art.

Nous traitons également de possibles optimisations guidées par les requêtes afin d'améliorer les architectures des moteurs de recherche Web. A la différence des systèmes pair-à-pair, les moteurs de recherche pour le Web utilisent actuellement des mécanismes de cache centralisés pour réduire la charge de requêtage de l'index principal. Dans cette perspective, nous analysons les historiques de requêtes ("query logs") de plusieurs moteurs de recherche existants pour

montrer que les modifications induites dans le trafic de requêtage par de telles techniques ont un impact sensible sur les performances de l'indexation. En particulier, nous étudions l'impact de ces modifications sur la performance de l'élagage de l'index. Nous montrons que la combinaison des deux techniques apporte une réduction importante des coûts de traitement des requêtes, et, de fait, présente un intérêt pratique pour les moteurs de recherche Web.

Mots clefs: systèmes de grande taille, indexation guidée par les requêtes, P2P, P2PIR, recherche d'information, XML, moteur de recherche Web, cache, élagage d'index.

Acknowledgements

The first person I would like to thank is my thesis supervisor, Prof. Karl Aberer. He did an excellent job in supporting me throughout my PhD and at the same time gave me the opportunity to concentrate on the topics I liked most. I learnt a lot from Karl and I consider myself very lucky that I did my PhD in his lab.

I am very thankful to the members of my thesis committee: Prof Anastasia Ailamaki, Prof. Ricardo Baeza-Yates, Prof. Monika Henzinger and Dr. Fabrizio Silvestri for their important comments and discussions to improve my dissertation.

I wish to thank all my colleagues whom I collaborated with during the work on this thesis, especially Manfred Hauswirth, Ivana Podnar Žarko, Martin Rajman, Toan Luu and Sebastian Michel.

I thank all my colleagues from the laboratoire de systèmes d'information répartis – we have a great team. A special thanks goes to Chantal who helped me sort out so many not only administrative issues.

I also want to thank the Yahoo! Research Barcelona group for the wonderful 4 months I spent there during my internship. I learned many interesting things there and was lucky to work with Flavio Junqueira, Vassilis Plachouras and Ricardo Baeza-Yates.

I thank all my friends from the doctoral school group and beyond, for their support and for all the great moments we spent together, including all our travels, mountain adventures and parties: Ivana, Šarūnas, Kasia, Michal, Marta, Maciej, Mallory, Maxim, Ivana, Dan, Parisa, Wojtek, Irina, Valya, Alexei, Iuli, Oana, Razvan, Carla, Radu, Marcin, Yura, Natasha, Denis, Philippe, Roman, Sebastian, Sheila, Claudia, Simon and many many others.

Finally, I want to thank Adriana, Sasha, Zoya and my parents Elena and Kirill for their support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> If we want things to stay as they are,
> things will have to change.
>
> Giuseppe Tomasi di Lampedusa

1.1. Searching Large-Scale Data Repositories

We have witnessed an exponential growth of the amount of Web content in the past twenty years since the beginning of the World Wide Web in 1989. As for 2008, recent studies report almost 30 Billion Web pages[1] on the *surface Web* – a part of the Web indexed by search engines, more than 500 Million Internet domains[2] and about 1.5 Billion Internet users[3]. The size of Web data has reached the order of petabytes and is constantly growing. Furthermore, according to [Bergman 2001] the amount of the Web content that is not indexed by search engines, called the *deep Web*, could be 400-550 times larger than the surface Web.

It is no surprise that searching in this ocean of data poses serious challenges in terms of quality and performance. Web search engines such as *Google*[4] and *Yahoo!*[5] rely upon large complex systems using thousands of servers, interconnected through different networks, and often spanning multiple data centers in order to be able to handle thousands of queries per second[6]. To sustain such loads and support sub-second query processing times, search engines constantly crawl the surface Web and build an index over the fetched data. The index is used for query processing and is distributed and replicated on many servers. At such a scale all components of the system have to be carefully optimized and many sophisticated techniques that reduce the resource utilization are employed, in particular *caching*.

[1] http://www.worldwidewebsize.com
[2] http://www.isc.org/ops/ds
[3] http://www.internetworldstats.com/stats.htm
[4] http://www.google.com
[5] http://www.yahoo.com
[6] http://www.comscore.com/press/release.asp?press=2230

Nowadays, commercial search engines earn most of their revenue by embedding context-sensitive advertisements in the search results. This search monetization model enables substantial investments in the infrastructure that permits search engines to cope with constantly increasing loads and deliver reliable service. Thus, for most of the Internet users Web search became a basic service in many aspects of their everyday life.

However, despite the general success of Web search, the problem of efficient and effective searching in large-scale data repositories is far from being solved. For instance, improving search accuracy is at the current focus of research. The Web comprises various types of content in the form of textual documents, structured metadata, databases, maps, images, video, *etc.* Using specific properties of each type could potentially increase the quality of search but requires non-trivial solutions to handle large volumes of data. In particular, structural information found on the Web is often ignored at indexing time thereby potentially harming search accuracy. Other search quality related issues include relevance computation, data freshness, spam detection, duplicate page removal, language detection, *etc.*

But possibly the main challenge is *scalability* – the ability to cope with the growing amount of information and the increasing demand for the search service. A recent survey by [Baeza-Yates *et al.* 2007a] envisions that the number of servers required by a search engine to keep up with the load in 2010 might reach the order of millions. This could be unfeasible for the cluster-based architecture currently employed by commercial search engines. Thus, it is important to design a truly distributed large-scale system that enables fast and accurate search over very large amounts of content.

In this thesis we study novel indexing strategies that enable efficient search in distributed large-scale data repositories. In particular, we introduce and explore the concept of *query-driven indexing* – an index construction strategy that adapts to the querying patterns expressed by users. The idea is to abandon the strict difference between indexing and caching, and to build a distributed indexing structure, which is optimized for the current query load.

We show that employing query-driven indexing is especially beneficial when the content is (geographically) distributed in a Peer-to-Peer network. In such a setting extensive bandwidth consumption has been identified as one of the major obstacles for efficient large-scale search. Our indexing mechanisms combat this problem by maintaining popularity statistics and indexing (caching) intermediate query results that are requested frequently. We present several indexing strategies for processing multi-keyword and XPath queries over distributed collections of textual and XML documents respectively.

Contrary to the Peer-to-Peer setting, Web search engines use centralized caching of query results to reduce the processing load on the main index. We analyze real query logs and show that the changes in the query traffic that such a results cache induces fundamentally affect indexing performance. In particular, we study its impact on the efficiency of (static) index pruning – a technique that reduces the resource utizilation by employing a smaller version of the main index for query processing. We consider query-driven techniques to construct such a pruned index.

1.1.1. Peer-to-Peer Networks

Peer-to-Peer (*P2P*) systems have been very successful as global-scale file-sharing systems (*e.g.*, BitTorrent[7]). A recent study by Sandvine Inc. reports that up to 44% of the bandwidth in North America is consumed by P2P file sharing [Sandvine Inc. 2008]. Despite the fact that the success mainly comes from the illegal content distribution, Peer-to-Peer networks have been also intensively studied in the research community in the past years. Nevertheless, there are only few applications except for file sharing where Peer-to-Peer systems are being practically deployed, for example: Seti@home[8] for distributed computation, Skype[9] for IP telephony and Zatoo[10] for video streaming. However, it is clear that with a right "killer" application a P2P network could easily reach millions of users in a short time, *e.g.*, Skype recently reported 12 Million users concurrently online[11].

1.1.2. Peer-to-Peer Data Management

To make Peer-to-Peer systems a viable architectural alternative for more technical and database-oriented applications than simple file sharing, support for powerful and expressive queries is required. A couple of approaches have been suggested already on top of unstructured P2P systems, for example Edutella [Nejdl *et al.* 2002]. Unstructured P2P systems do not use indexing, but typically employ some form of constrained flooding to locate resources. Thus, they can handle queries of arbitrary complexity, since each peer receiving the query can locally evaluate it and return its contribution to the overall result set. However, this comes at the expense of very high bandwidth consumption and some intrinsic limitations.

The efficient alternative are structured P2P systems, as they typically offer logarithmic search complexity in the number of participating nodes. Although the use of a distributed index (typically a Distributed Hash Table) enables more efficient query processing, it also introduces considerable complexity in an environment that is as instable and error-prone as large-scale Peer-to-Peer systems.

Since the first structured Peer-to-Peer networks[12] appeared in 2001, designing a large-scale data storage on top of such systems became a popular research topic. Various approaches implementing P2P relational databases (*e.g.*, [Harren *et al.* 2002]), P2P XML management systems (*e.g.* [Abiteboul *et al.* 2008]) and P2P Information Retrieval systems (*e.g.* [Bender *et al.* 2005b]) were proposed. In particular, P2P XML (or more recently RDF) management became a popular research direction in the context of distributed digital libraries.

[7] `http://www.bittorent.com`
[8] `http://setiathome.berkeley.edu`
[9] `http://www.skype.com`
[10] `http://zattoo.com`
[11] `http://share.skype.com/sites/en/news_events_milestones`
[12] *E.g.*, Chord [Stoica *et al.* 2001], Pastry [Rowstron and Druschel 2001], CAN [Ratnasamy *et al.* 2001] and P-Grid [Aberer 2001].

Many approaches implementing a P2P data storage rely on some sort of distributed indexing structure optimized for processing conjunctive queries – a crucial functionality of a large-scale data repository. Standard techniques for processing conjunctive queries in P2P systems are query flooding and distributed inverted list intersection. However, it has been shown that both methods fail to scale well with the size of the data. Thus, a number of optimizations have been suggested recently based on these techniques. In Chapter 3 we present an approach for XPath query processing in a P2P XML data storage that employs a dedicated caching strategy in order to avoid expensive query flooding.

1.1.3. P2P Information Retrieval

In reaction to the scalability problems encountered with centralized information retrieval engines, P2P networks that distribute a global index over a large number of interconnected peers may be considered as a promising solution to cope with Web-scale document retrieval. While P2P networks containing very large number of peers indeed provide virtually unlimited storage capacities, there is no evidence about true scalability of P2P Web search. Particularly, [Li *et al.* 2003] show that a naïve use of structured or unstructured P2P networks for Web retrieval leads to unscalable network traffic, and, even for more sophisticated schemes, such as term-to-peer indexing [Bender *et al.* 2005a; Cuenca-Acuna *et al.* 2003] or hierarchical federated architectures [Balke *et al.* 2005b; Lu and Callan 2005], only little evidence of their scalability is available. In fact, [Zhang and Suel 2005] have shown that, even when carefully optimized, distributed algorithms using traditional single-term indices in structured P2P networks generate unscalable network traffic during retrieval, mainly because of the bandwidth consumption resulting from the large posting list intersections required to process queries with frequent terms. In Chapters 4 and 5 we propose query-driven indexing techniques for medium-scale and Web-scale full text retrieval in Peer-to-Peer networks that efficiently combat the problem of excessive traffic consumption.

There is an ongoing debate on the practical feasibility of large-scale Peer-to-Peer information retrieval. From the technological point of view, there is nothing a Peer-to-Peer system can offer, which can not be implemented with a centralized system provided that it has sufficient resources. Even with its main advantage – a possibility for no-investment bootstrap, Peer-to-Peer systems can hardly compete with commercial search engines, which can invest their advertising revenues in the infrastructure. Especially when search engine users are used to nearly instant query processing times backed up by huge data centers, P2P systems nowadays can not impress in terms of query latency.

Nevertheless, while commercial Web search engines are looking towards more distributed architectures to cope with the scalability challenge [Baeza-Yates *et al.* 2007a], it could happen that some techniques developed in the P2P-IR domain would be adopted by search engines in the near future or new alternative ways of searching on the Web will appear. For example

several P2P-IR startups such as Faroo[13] and YaCy[14] have appeared recently. Another example is the AlvisP2P research prototype[15] developed in our group at EPFL (see Chapter 5).

Apart from the scientific interest in large-scale distributed architectures for P2P-IR, we identify several reasons why P2P Web search can be of practical interest and could possibly successfully complement commercial search engines in the future:

Queries from the long-tail: [Broder 2002] classifies Web queries into three classes: *navigational* (the intent to reach a particular site), *transactional* (the intent to perform some Web-mediated activity) and *informational* (the intent to acquire some information assumed to be present on one or more web pages). Given the limited resources, search engines would rather optimize the performance of navigational and transactional queries because they better contribute to their advertising model. On the other hand, a P2P-IR system might potentially deliver better results for informational queries than traditional centralized Web search engines due to a different incentive mechanism.

Heterogeneity: Web search engines use pull mechanisms to populate their indices: they periodically crawl the Internet, extract textual elements from the acquired Web pages and build their indexes from these elements. Such uniform and centralized processing implies that some specific indexing features (e.g., complex gene names in bioinformatic collections, or formulas in math or chemistry related sites) might get lost unless they are specifically supported by the search engine, which requires a substantial centralized effort. In contrast, P2P-IR systems would naturally use push mechanisms: peers decide themselves which documents they want to make globally searchable and, more importantly, how these documents should be indexed. Thus, the effort of handling heterogenous data is distributed in the network and can be managed more efficiently. Such a scenario is therefore appropriate for the management of heterogeneous, frequently changing document collections. For instance, a specialized digital library could continue to use its own sophisticated means to index/query local documents, while using a P2P-based IR infrastructure as a common search framework that makes its specialized indexing/retrieval means available to the whole P2P network.

Provider-centric vs. broker-centric approach: Major search engines play a central role as information brokers, but not so much as information creators. Therefore, IR infrastructures allowing novel business models based on the direct rewarding of original content providers might be considered. Within this perspective, P2P-IR systems exhibit interesting characteristics: namely, as already mentioned, in such systems only the document indexes are published in the network, while the documents (and thus the associated added value) always remain under the control of the original provider (peer). The peers can therefore decide about the conditions upon which the document access will be granted (*e.g.*, free access, micro-payment, subscription, remunerated advertizement, *etc.*).

[13] `http://www.faroo.com`
[14] `http://yacy.net`
[15] `http://globalcomputing.epfl.ch/alvis`

Community-based search: A P2P-IR client is an easy to install software that requires only limited resources from the hosting system. A P2P-IR network is therefore quite simple to deploy, as it does not require anyone taking the responsibility of setting up a centralized server (or network of servers). Consequently, as soon as P2P-IR clients become widely available, building topical communities sharing a document collection within a given domain should become simple. Thus, as the emergence of such topical communities in fact corresponds to an implicit structuring of the global Web-scale document collection, this opens an interesting possibility to fight the unavoidable precision drop associated with the growth of any document collection.

1.2. Contribution of the Work

In this thesis we make the following contributions to indexing and caching techniques for search optimization in large-scale information systems:

- In the area of Peer-to-Peer Information Retrieval (Chapters 4 and 5):
  - We introduce the *Distributed Cache Table* (DCT) approach (Chapter 4) – a decentralized cache of query results deployed in a structured Peer-to-Peer network for medium-scale text retrieval.
  - We study how subsumption of multi-term queries can be used for cache selection in the DCT approach and perform extensive experiments with real Wikipedia[16] query logs that show a significant overall traffic reduction compared to the distributed single-term indexing approach.
  - We introduce an alternative *Query-Driven Indexing* (QDI) strategy (Chapter 5) for *Web-scale* Peer-to-Peer text retrieval that guarantees scalable storage/bandwidth requirements and is based on: 1) indexing of popular term combinations, and 2) truncating the posting lists containing too many document references.
  - We provide a scalability analysis of the QDI approach, based both on theoretical results and experimental evaluations, that shows the viability of our approach for Web-scale document collections.

- In the area of Web search engine architectures (Chapter 6):
  - We show that centralized results caching plays a crucial role in optimizing query processing for Web search engines. It guarantees high cache hit rates with a constant cache capacity independently of the document collection size – an important advantage compared to index pruning for example.

[16] `http://www.wikipedia.org`

- We compare the properties of the original query stream and the stream of misses after the results cache, and show how the differences affect the applicability of index pruning.
- We compare the efficiency of various static index pruning techniques when a pruned index is used separately or in combination with a results cache.
- We propose a different technique for combining term and document pruning that outperforms state-of-the-art methods.

- In the area of Peer-to-Peer XML data management (Chapter 3):
  - We present an indexing strategy that is optimized for processing XPath queries in a structured Peer-to-Peer network.
  - We introduce caching of intermediate results to efficiently handle some XPath constructs and minimize the number of expensive multicasts.

1.3. Structure of the Thesis

The thesis is organized in the following way:

Chapter 2 gives an overview of the existing literature in the areas of P2P networks, distributed indexing, distributed information retrieval and search engine architectures. The remaining chapters follow the *chronological order*.

Chapter 3 presents an approach that aims at efficient processing of XPath queries over a large XML repository distributed in a structured P2P network. We suggest an indexing structure that is optimized for processing XPath queries and extensively employs *caching* of partial results to efficiently handle some XPath constructs. Because of the dedicated caching mechanism, the content of the index is directly influenced by the query load. Thus, we can already refer to such an indexing strategy as *query-driven*. However, the concept of query-driven indexing is exploited in greater detail in the next two chapters.

In **Chapter 4** we continue to explore query-driven indexing strategies for distributed query processing in structured P2P networks, now applied to the P2P text retrieval scenario. We propose an approach for efficient processing of *multi-term* queries over a large collection of textual documents distributed in a P2P network. A query is resolved either 1) by locating a peer that can process the query locally using previously cached results, or 2) by broadcasting the query to the whole network. A distributed index is built on top of the underlying DHT to efficiently locate the caches that can answer a given multi-term query, thus, minimizing the number of expensive broadcasts. By analogy with Distributed Hash Tables (DHTs) we call this approach *Distributed Cache Table* (DCT). Our experimental results show up to two orders of magnitude traffic reduction at retrieval compared to the standard single-term indexing approach. However, while being suitable for middle-scale distributed digital libraries, DCT would be impractical for Web-scale document collections due to expensive cache maintenance.

In **Chapter 5** we further explore query-driven indexing methods for Peer-to-Peer text retrieval and propose a new *Query-Driven Indexing* (QDI) strategy. The QDI approach extends the Distributed Cache Table mentioned above and the HDK approach [Lu 2007], outlined in Section 2.2, to enable *Web-scale* text retrieval. It is based on two important properties: (1) the generated distributed index stores posting lists for *carefully chosen indexing term combinations* that are frequently present in user queries, and (2) the posting lists containing too many document references are truncated to a *bounded number of their top-ranked elements*. These two properties guarantee acceptable latency and bandwidth requirements, essentially because the number of indexing term combinations remains scalable and the posting lists transmitted during retrieval never exceed a constant size. An index update mechanism efficiently handles adding of new documents to the document collection. Thus, the generated distributed index constantly evolves by following current information needs of the users and changes in the document collection.

Finally in **Chapter 6** we switch our attention from the Peer-to-Peer index organizations to the classical Web search engine architecture with the purpose of investigating the advantages of query-driven indexing in this setting. In particular, we explore the results caching and index pruning techniques that employ information about past queries for query processing optimization. We propose *ResIn* – a variation of the Web search engine architecture that combines results caching and static index pruning. We study the performance of both components separately and investigate how they affect each other in a realistic setting.

We provide the conclusions of our work and outline future directions in **Chapter 7**.

1.4. Selected Publications

This thesis is based on the following main publications:

Peer-to-Peer XML data management

- Gleb Skobeltsyn, Manfred Hauswirth, Karl Aberer: *Efficient Processing of XPath Queries with Structured Overlay Networks.* Proceedings of the International Conference on Ontologies, Databases and Applications of SEmantics (ODBASE), October 31 – November 4, 2005, Agia Napa, Cyprus.

Peer-to-Peer information retrieval

- Gleb Skobeltsyn, Karl Aberer: *Distributed Cache Table: Efficient Query-Driven Processing of Multi-Term Queries in P2P Networks.* Proceedings of the CIKM Workshop on Information Retrieval in Peer-to-Peer Networks (P2PIR), November 11, 2006, Arlington, USA.

- Gleb Skobeltsyn, Toan Luu, Ivana Podnar Žarko, Martin Rajman, Karl Aberer: *Web Text Retrieval with a P2P Query-Driven Index.* Proceedings of the 30st International ACM SIGIR Conference, July 23 – 27, 2007, Amsterdam, The Netherlands.

- Toan Luu, Gleb Skobeltsyn, Fabius Klemm, Maroje Puh, Ivana Podnar Žarko, Martin Rajman, Karl Aberer: *AlvisP2P: Scalable Peer-to-Peer Text Retrieval in a Structured P2P Network (demo paper).* Proceedings of the 34th International Conference on Very Large Data Bases (VLDB), August 24 – 30, 2008, Auckland, New Zealand.

- Gleb Skobeltsyn, Toan Luu, Ivana Podnar Žarko, Martin Rajman, Karl Aberer: *Query-Driven Indexing for Scalable Peer-to-Peer Text Retrieval.* Future Generation Computer Systems, Volume 25, Issue 1, January, 2009.

Web search engine architectures

- Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, Ricardo Baeza-Yates: *ResIn: A Combination of Result Caching and Index Pruning for High-performance Web Search Engines.* Proceedings of the 31st International ACM SIGIR Conference, July 20 – 24, 2008, Singapore.

Chapter 2

State of the Art

In this section we summarize the related work relevant to the scope of the thesis. We first cover major topics in the area of Peer-to-Peer networks in Section 2.1 including structured/unstructured P2P organizations, Distributed Hash Tables (Section 2.1.1) and P2P data indexing (Section 2.1.2). We specifically look into XML management in P2P networks in Section 2.1.2.2.

We then switch our attention to distributed Information Retrieval (distributed IR) in Section 2.2. We first look into index partitioning principles in Section 2.2.1 and discuss architectures of Web search engines with an emphasis on caching and index pruning in Section 2.2.2. We then provide the overview of P2P Information Retrieval in Section 2.2.3 specifically focusing on the HDK approach [Lu 2007; Podnar *et al.* 2007], which we extend with query-driven indexing in Chapter 5.

2.1. Peer-to-Peer Networks

Nowadays most of the popular Web applications such as search engines, mail services, file hosting, photo sharing, social networks, *etc.* rely on centralized architectures. In order to handle high workloads they are deployed on a large number of servers possibly spanning several data centers but require a certain degree of centralized administration and control. P2P networks, on the other hand, target a different niche of applications, where each peer is autonomous and no central coordination is imposed. In this section we describe the classification of P2P approaches and present major P2P systems.

Simple file-sharing applications such as Napster[1] and Gnutella[2] made the Peer-to-Peer idea popular. Generally, a P2P network is an *overlay network* that consists of a number of peer nodes (*peers*) that act both as clients and servers ("servents") at the same time. The term

[1] `http://en.wikipedia.org/wiki/Napster`
[2] `http://en.wikipedia.org/wiki/Gnutella`

overlay is used to emphasize that the network is built on top of another network, for example on top of the Internet. The main properties of the Peer-to-Peer paradigm are:

- Every participating node acts both as a client and a server ("servent");
- Every node contributes by providing access to (some of) its resources;
- No central coordination;
- No central database;
- Peers are autonomous;
- No peer has a global view of the system;
- Global behavior emerges from local interactions;
- All existing data and services are usually accessible from any peer;
- Peers and connections are often unreliable.

While the number of peers in a P2P network could potentially be very large – locating the peer or the set of peers that hold a specific resource is a challenging problem. Various solutions for efficient resource lookup that rely on some form of indexing have been suggested.

To evaluate the efficiency of P2P systems, usually the communication overhead in terms of network messages is used as the dominating cost factor. Some P2P systems exhibit logarithmic *search cost*, *i.e.*, the number of messages required to locate a given resource grows logarithmically with the number of peers.

P2P systems can be classified as *centralized*, *decentralized* and *hybrid*. Centralized systems such as Napster use a central directory for the index – while resources (music files for Napster) are stored on the individual peers, each resource lookup has to go through the central sever. This solution does not scale because one node gets all index traffic and is a single point of failure. Shutting down the central server causes the whole network to stop operating, which is exactly what happened to Napster in 2001 due to copyright issues.

Decentralized P2P systems rely on distributed index structures, thus avoiding a single point of failure. Each peer plays an equal role maintaining only a fraction of the index. Alternatively, in hybrid systems, some peers (called super-peers) play the role of a "centralized" server for a small subset of "normal" peers.

P2P systems can also be classified as *unstructured* and *structured* based on their structural properties (*i.e.*, topology) and the level of binding and dependency among the peers (*i.e.*, how much "knowledge" peers maintain of other peers). In **unstructured P2P systems** such as Gnutella[3] peers organize according to an arbitrary topology. This causes overheads in search efficiency, but on the other hand makes the system flexible and robust. There exist different

[3] `http://en.wikipedia.org/wiki/Gnutella`

searching methods in unstructured systems, which are usually based on a constrained broadcast approach or random walks [Chawathe *et al.* 2003]. Constrained broadcast forwards a message to all neighboring peers, which in turn forward the message until its Time-To-Live (TTL) counter expires. This solution causes a significant amount of redundant traffic. A random walk is based on randomly transmitting one (or more) messages (walkers) between peers until the particular element is found. The walker periodically asks the originator of the request – if it should continue the search or not. Such walkers are not efficient in terms of time required to perform the search operation.

Structured P2P systems, on the other hand, have a strict topology and each node stores and maintains information about few other peers to support efficient routing. The search cost is reduced noticeably comparing to unstructured systems, but additional maintenance has to be performed. For example, joining and leaving nodes have to be considered (update of the routing tables, redundant entries) and also changes in the structure of the network have to be taken into account. If the population of nodes in the network does not change drastically over time, structured systems typically perform much better than unstructured systems in terms of bandwidth consumption. The majority of structured P2P networks implement the hash table interface and are therefore called Distributed Hash Tables (DHTs).

2.1.1. Distributed Hash Tables (DHTs)

A *distributed hash table* (DHT) is a hash table whose entries are distributed among different peers. Each peer implements a variation of a so-called *put/get* interface inherited from hash tables: 1) a resource can be placed in the network using the *put(resource)* function, and 2) a resource can be retrieved from the network using the *get(key)* function. As the name suggests, DHTs use hash functions to compute the key that is used to identify the peer responsible for a resource. Usually locating this peer is efficient in terms of overlay hops and for the majority of approaches it takes $O(\log N)$ hops to route to any peer in the network, where N is the number of peers.

Each peer in a DHT is responsible for a partition of the overall key space and maintains a *routing table* such that it can forward requests that cannot be answered locally to other peers (neighbors). The routing table of a peer is constructed such that it holds peers with exponentially increasing distance in the key space from its own position in the key space. This technique basically builds a small-world graph [Kleinberg 2000], which enables search in $O(\log N)$ steps. Essentially, all systems referred to as DHTs are based on variations of this approach and only differ with respect to fixed vs. variable key space partitioning, the topology of the key space (ring, interval, torus, *etc.*), and how the routing information is maintained (redundant entries, dealing with network dynamics and failures, *etc.*). Routing tables are always constructed in such a way that they cover the whole key space (completeness property). This means a request for a certain key can be routed to the responsible peer starting from any peer in the network.

Popular DHT implementations include CAN [Ratnasamy *et al.* 2001], Chord [Stoica *et al.*

2001], Pastry [Rowstron and Druschel 2001], Kademlia [Maymounkov and Mazières 2002] and P-Grid [Aberer 2001]. We briefly describe them in the next section.

2.1.1.1. DHT Implementations

CAN [Ratnasamy *et al.* 2001] uses a d-dimensional Cartesian coordinate space. The coordinate space is partitioned into hyper-rectangles. Each node is responsible for one hyper-rectangle, also called zone. In CAN, a node is identified by the boundaries of the zone it is responsible for. Two nodes are neighbors if their zones share a *d-1* dimensional hyper-plane. Thus, the routing table for each peer contains links to its neighbors.

Routing in CAN is done by forwarding a message along the path that approximates the straight line in the coordinate space from the originator of the message to the node responsible for the key. The size of routing table is $O(d)$ and the routing cost is $O(dN^{1/d})$. Assuming d is $O(\log N)$ the routing cost would be $O(\log N)$, thus exhibiting the "standard" DHT performance.

To join the network, a node chooses an arbitrary point in the key space and contacts the node responsible for this point by asking any peer in the network. The requested node splits its zone into two equal parts and assigns one of the halves to the new node. If a node leaves, its zone is taken over by a neighbor. A dedicated algorithm handles node failures.

Chord [Stoica *et al.* 2001] uses a one-dimensional key space for both keys and node addresses (IDs). The key space forms a ring, so that ID 0 follows the highest ID. The node whose ID most closely follows a key k is responsible for it and is called the successor of k. Chord's routing tables, called *finger tables*, contain the links to the nodes that are halfway around the ID space from a specific node, a quarter, one eighth and so forth until the immediate successor. To ensure correct and robust routing, each node additionally keeps a set of links to the next r immediate successors. This list is called *successor list*.

Routing in Chord is done by greedily approaching the node that is responsible for a given key using the finger tables. A node forwards the message to one of the nodes from its finger table whose ID is highest but not greater then the destination key. The power-of-two structure of the finger table ensures $O(\log N)$ routing cost.

To join the network, a node chooses an arbitrary ID and finds the peer responsible for this ID. The new node and existing nodes have to update their finger tables.

Pastry [Rowstron and Druschel 2001] nodes randomly choose 128-bit IDs in the base 2^b, where b is usually 4. As in Chord, the key space is also represented by a ring, and the peer with the ID numerically closest to a key is responsible for that key. Pastry uses a prefix-based forwarding scheme. Each node A stores a leaf set L, which consists of: 1) the set of $|L|/2$ nodes whose IDs are the closest to and smaller then A's ID, and 2) the set of $|L|/2$ nodes whose IDs are the closest to and larger then A's ID. This leaf set guarantees correct routing and is conceptually similar to Chord's successor list. The routing table contains links to the other

peers in the ID space and the number of rows in the routing table is $\log_{2^b} N$. The i-th row points to a set of nodes whose IDs start with the same prefix of length i as the current node with the $i+1$th digit different, thus, $2b - 1$ links are stored in each row.

While routing to the node responsible for a key k, node A uses its leaf set and the routing table. If k is covered by the leaf set, the message is forwarded to the node responsible for k. If not, the message is forwarded to the node from the routing table that has a longer shared prefix with k then A. If such a node does not exist, the message is forwarded to the node with the same shared prefix as A, but which is numerically closer to k. The upper bound of routing cost is $\log_{2^b} N$. The routing algorithms of Tapestry [Zhao *et al.* 2001] are similar to Pastry.

P-Grid [Aberer 2001] is a tree-based DHT that was designed at EPFL. P-Grid uses a virtual tree to position peers in the ID space. Routing tables of $O(\log N)$ size enable efficient lookup with $O(\log N)$ messaging cost. Contrary to other DHTs, P-Grid uses an order-preserving hash function that provides a natural support for range queries [Datta *et al.* 2005]. For this reason we use P-Grid to design a P2P XML storage in Chapter 3 and provide a more detailed description of it in Section 3.2.

Kademlia [Maymounkov and Mazières 2002] is a symmetrical DHT-based overlay that uses a XOR-based distance metric to construct its topology and assign resource advertisements to peers. Kademlia's symmetrical architecture enables the usage of query messages for maintenance purposes, thus reducing the maintenance costs. Kademlia allows peers to select their neighbors from sets of peers sharing the same prefix.

Other approaches. [Girdzijauskas *et al.* 2005] show that the logarithmic-style DHT approaches described above form topologies that follow Kleinberg's small world principles [Kleinberg 2000]. They belong to the special class of "routing efficient" small-world networks where decentralized, greedy search algorithms provide the best performance. Therefore, conceptually, all these approaches build similar small-world graphs with certain constraints for each case. Symphony [Manku *et al.* 2003] follows Kleinberg's principle while constructing the peer's routing tables resulting into a small-world topology. For other logarithmic-style overlay networks, each peer views the identifer space as split in $\log N$ partitions where each partition is b times bigger than the previous one (b is the radix of the identifer alphabet). The peers' routing tables in such systems contain $\log_b N$ links to some nodes from each partition. In Chord the chosen node will be the one with the smallest identifer of the given partition, while Pastry and P-Grid use any random node in the partition, which is a more relaxed constraint.

[Aberer *et al.* 2005] identify common properties of P2P approaches and presents a reference architecture for overlay networks including a qualitative comparison and standardized API principles. Another good survey is published by [Lua *et al.* 2005].

2.1.2. Indexing in Structured Peer-to-Peer Networks

In the context of databases, indexing is a technique used by all current database management systems to speed up the execution of particular kinds of queries. A dedicated redundant data

structure is typically maintained such that requested table entries can be located quicker as opposed to a sequential scan through the whole table. The query processing speedup due to indexing becomes even more significant when the database is distributed.

Peer-to-peer systems can be viewed as a form of loosely-coupled distributed databases with limited (or without any) central coordination and knowledge. Data lookups in Peer-to-Peer networks are usually done in two steps: 1) finding the node(s) storing the requested resource(s), and then 2) retrieving the resource(s) locally at each node and return them to the user. Very often only the first step is at the focus of Peer-to-Peer research because the latency caused by local lookup can be often neglected compared to the network delays during the peer lookup.

Structured Peer-to-Peer systems rely on indexing to enable efficient lookup. Various P2P indexing approaches exist depending on the types of queries they support: conjunctive queries, range queries, similarity queries, structured queries, *etc.* Many of them rely on the basic DHT indexing functionality and can potentially be deployed on top of any DHT system. Some might rely on specific properties of a concrete P2P approach or even design a dedicated overlay network.

2.1.2.1. Peer-to-Peer Data Indexing Approaches

The use of routing indices [Crespo and Garcia-Molina 2002a] facilitates the construction of a P2P network based on content. In such content-based overlay networks peers are linked if they keep similar data, *i.e.*, each peer maintains summaries of the information stored at its neighbors. While searching, a peer uses the summaries to determine whom to forward a query to. The idea of clustering peers with semantically close content is exploited by [Crespo and Garcia-Molina 2002b].

The Edutella project [Nejdl *et al.* 2002] is a P2P system based on a super-peer architecture, where super-peers are arranged in a hypercube topology. This topology guarantees that each node is queried exactly once for each query, which presumes powerful querying facilities including structured queries, but does not scale well.

[Datta *et al.* 2005] leverage P-Grid [Aberer 2001] to implement range queries. Two range query processing algorithms are proposed: min-max traversal and the shower algorithm. Both rely on the key-locality property of P-Grid: keys that share the same binary prefix are stored close to each other because of the order-preserving hash function used in P-Grid. Thus, the set of peers storing keys matching a given range are neighbors in the overlay structure and can be efficiently contacted using a constrained multicast. We use the shower broadcast algorithm in Chapter 3 for XPath query processing.

[Kothari *et al.* 2003] propose a range addressable network architecture that facilitates range query lookups by storing the query results in a cache. [Sahin *et al.* 2004] leverage the CAN P2P network to address a similar problem. In both cases, queries are specified as integer intervals. The ranges themselves are hashed, which makes simple key search operations inefficient.

[Ganesan *et al.* 2004] describe indexing of multidimensional data in the P2P network using skip graphs [Aspnes and Shah 2003]. The authors propose two approaches: SCRAP – mapping of multidimensional data into a single dimension using space-filling curves, and MURK – a multidimensional extension of skip graphs. Space-filling curves are also used by [Chawathe *et al.* 2005] who suggest using Prefix Hash Trees – a data structure for geographic range queries that is built on top of a standard DHT.

MAAN [Cai *et al.* 2004] extends Chord to answer multi-attribute queries in the grid environment. Ranges are supported using a locality-preserving hash function. MAAN uses a single attribute dominated approach, *i.e.*, one (the most selective) attribute-value pair is used to route to the peer that answers the whole query. Thus, data records are replicated across the network for each attribute. We employ a similar approach in Chapter 4 while caching document digests. RDFPeers [Cai and Frank 2004] use the MAAN architecture for indexing RDF triples.

Mercury [Bharambe *et al.* 2004] is an overlay network designed to support multiple-attribute range queries. The network consists of several sub-networks (hubs) – each responsible for one attribute. These sub-networks are organized in order-preserving circular overlays such that each peer is responsible for a certain range of a certain attribute. In order to handle skewed value distributions Mercury performs data sampling.

Oscar [Girdzijauskas *et al.* 2006, 2007] is an overlay network based on small world graphs. Oscar efficiently handles the bandwidth/storage heterogeneity of peers, as well as the non-uniformity observed in data-oriented applications, *i.e.*, skewed key distributions and skewed workloads. Oscar is similar to P-Grid [Aberer 2001] as it supports complex non-uniform key distributions, but does not suffer from node in-degree imbalance while exhibiting good lookup performance. Unlike Mercury, the construction of the Oscar overlay does not require an approximation of the key skew over the entire key space. Oscar employs a scalable sampling technique, where only very few samples of the network are needed to construct routing efficient networks given key distributions of any complexity. Thus, Oscar supports range queries over very skewed distributions.

A decentralized P2P cache of Web pages called Squirrel is proposed by [Iyer *et al.* 2002]. Squirrel enables Web browsers on desktop machines to share their local caches in a Pastry DHT network, to form a decentralized (*e.g.*, corporate) Web cache.

Finally, the PIER project [Harren *et al.* 2002; Huebsch *et al.* 2005] implements a distributed execution of relational database query operations on top of a DHT including joins and aggregation queries.

In Chapter 3 we propose an approach for XPath query processing in structured P2P networks, hence we devote the next section to provide an overview of indexing approaches specifically targeted to XML management in structured P2P networks. Another important cluster of P2P indexing approaches – P2P full text retrieval is discussed in Section 2.2.3.

2.1.2.2. Peer-to-Peer XML Management

Many approaches exist that deal with querying of XML data in a local setting. Most of them try to improve the query-answering performance by designing an indexing structure for local data processing. Examples of such index structures include DataGuides [Goldman and Widom 1997], T-indexes [Milo and Suciu 1999], Index Fabric [Cooper *et al.* 2001], the Apex approach [Chung *et al.* 2002] and others. [Mandhani and Suciu 2005] propose caching of materialized views for query processing in a local XML database. The authors report up to 78% of queries answered using cached materialized views in their setup. However, these approaches are not targeted for a large-scale distributed XML storage.

On the other hand, Peer-to-Peer networks yield a promising solution for storing huge amounts of XML content. The important metrics of such systems are:

- Flexibility of the querying mechanism (*e.g.,* query language);
- Costs of query processing and maintenance.

[Koloniari and Pitoura 2004] propose using multi-level bloom filters [Bloom 1970] to summarize hierarchical data, *i.e.*, the similarity of peers' contents is based on the similarity of their filters. [Petrakis *et al.* 2004] use histograms as routing indexes and propose a decentralized procedure for clustering peers based on their histogram distances. A recent work by [Skyvalidas *et al.* 2007] focuses on replicating XML fragments in a P2P network using a data structure called Replication Routing Index. The content-based approaches could efficiently solve the problem of answering structured queries, though lack of structure affects the result set quality and significantly increases the search cost for large-scale networks.

[Galanis *et al.* 2003] index XML paths in a Chord-based DHT by using XML tag names as keys. A peer responsible for an XML tag stores and maintains a data summary with all possible unique paths leading to the tag. Thus, only one tag from a query is used to locate the responsible peer. Although ensuring high search speed, the approach introduces considerable overhead for popular tags, when the data summary is large. The paper also addresses answering branching XQuery expressions by joining the result sets obtained from different peers.

The approach by [Bonifati *et al.* 2004] also uses a Chord network, but follows a different technique. Path fragments are stored with the information about the super- and child-fragments. Having located a peer responsible for a path fragment, it resolves the query by navigating to the peers responsible for the descendant fragments. Additional information has to be stored and maintained to enable this navigation, which causes additional maintenance costs. For some types of queries the search operation may be rather expensive due to the additional navigation.

Caching is used by many P2P indexing approaches to improve performance. For example, [Garcés-Erice *et al.* 2004] utilize caching for efficient processing of XPath queries. Each descriptor – a top level segment in an XML file, is represented by the most specific query for

this segment, which includes all the data stored in the segment. The most specific queries are indexed in the DHT pointing to relevant documents. Less specific queries are also indexed pointing to more specific queries and so on. The querying process is recursive until the most specific queries are reached. The most popular queries are cached to boost the performance.

Cooperative caching of XPath queries was recently suggested by [Lillis and Pitoura 2008]. The authors exploit a prefix-based approach for caching results of path queries. Relevant to this thesis, their cache-based query processing relies on query subsumption. Two approaches are proposed: 1) in a loosely-coupled approach each peer stores results of its own queries in the local cache and just publishes the queries in the index, and 2) in a tightly-coupled approach each peer is assigned a specific part of the query space to cache. Cooperative caching is very similar to our DCT approach discussed in Chapter 4, however we concentrate on multi-keyword query processing. It also resembles the XPath indexing scheme we employ in Chapter 3.

A P2P XML retrieval system called SPIRIX is proposed by [Winter 2008]. SPIRIX relies on indexing with combinations of (content, structure)-tuples called XTerms extending the HDK approach [Luu 2007] described in Section 2.2.3.2. As a result, such an index can process "content-only" queries consisting of keywords as well as "content-and-structure" queries.

Finally, KadoP, proposed by [Abiteboul *et al.* 2008], indexes XML element labels (but not the paths) in a DHT and employs a Distributed Posting Partitioning (DPP) algorithm to split large posting lists among several peers. It also introduces structural Bloom filters to optimize the bandwidth consumption.

A good survey of P2P XML management approaches as of 2005 is published by [Koloniari and Pitoura 2005].

2.2. Distributed Information Retrieval

Broadly defined, Information Retrieval (IR) systems target matching of user queries to documents that are relevant to the queries. Relevance of a document to a query is explicitly modeled and usually only the top-k most relevant documents are returned. Typically, the majority of queries in such systems are multi-keyword queries. Documents might contain structured, semi-structured, or unstructured textual information. Full text retrieval is a branch of information retrieval that considers textual data only. In the common case of conjunctive query processing a text document is a valid answer for a query when it contains all terms from the query. For example, search engines imply AND semantics for multi-keyword queries by default: a query $\{t_1\ t_2\ ...\ t_n\}$ is the same as $\{t_1\ \&\ t_2\ \&\ ...\ \&\ t_n\}$.

To evaluate the quality of query processing, IR systems use various metrics with *precision* and *recall* being the most common ones. Precision and recall evaluate an IR system by comparing its query results to the ground truth usually provided by human experts. Formally,

$$precision = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|},$$

i.e., precision is the fraction of the retrieved results that are relevant. Formally,

$$recall = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{relevant\ documents\}|},$$

i.e., recall is the fraction of relevant results that are successfully retrieved.

Recall and precision are measures for the entire result list and do not account for the quality of ranking. In large-scale IR systems a query often returns many results and only the top-k of them are of interest. To capture the effect of ranking the *precision@k* measure is used. Precision@k is evaluated at a given cut-off rank, considering only the k topmost results returned by the IR system. Thus, *precision@k* is the proportion of the top-k documents judged relevant.

For efficient query processing IR systems rely on indexing, typically on a variation of the *inverted index* technique [Baeza-Yates and Ribeiro-Neto 1999]. The inverted index maintains a vocabulary – a list of all terms found in the document collection, and a number of *posting lists* for all terms from the vocabulary. A posting (or inverted) list of a term t stores the references to all documents that contain t together with some auxiliary information, *e.g.*, the statistics that are used for ranking (for example term frequency) or positional information which is used for resolving phrase queries. A multi-term query can then be processed by intersecting the posting lists of all query terms, computing (aggregating) the scores of the documents in the intersection and returning k documents with the highest scores.

2.2.1. Index Partitioning Principles

For large IR systems the inverted index becomes too big to fit on one server and has to be distributed. There are two main strategies for partitioning the index [Baeza-Yates and Ribeiro-Neto 1999; MacFarlane *et al.* 2000]: *term partitioning* and *document partitioning*. Figure 2.1 illustrates both strategies.

Document partitioning splits the document collection in several sub-collections and each sub-collection is indexed locally and independently on a different server. Thus, document partitioning is sometimes called local index partitioning. A query is processed by all servers in parallel and the final result is aggregated from the top-k local answers supplied by each server. Important advantages of document partitioning are the simplicity of the indexing procedure and nearly even load balancing between the servers. On the other hand, each query has to be processed by each server which increases the processing costs. Nevertheless, search engines employ document partitioning as being advantageous when deployed in a cluster with good network connectivity [Badue *et al.* 2001]. We will discuss Web search engine architectures and indexing principles in detail in Section 2.2.2.

Term partitioning assigns terms to servers such that each server maintains complete posting lists for certain terms. Hence, term partitioning is sometimes called global partitioning. To process a query only the servers responsible for the query terms have to be contacted.

However indexing is costly and it is hard to balance the load as the term frequency distribution follows a power law[4]. Intersecting posting lists that are stored on different servers can be time and bandwidth consuming. Nevertheless, Peer-to-Peer Information Retrieval (P2P-IR) approaches built on top of structured P2P networks (typically DHTs) often choose term partitioning, *e.g.* [Zhang and Suel 2005]. The main reason is that in large P2P networks it is important to restrict the query processing to a small number of peers instead of broadcasting each query to all peers as in the case of document partitioning. Also the put/get interface of DHTs can be easily extended to support such a term partitioned index. In Section 2.2.3 we discuss several P2P-IR approaches in more detail.

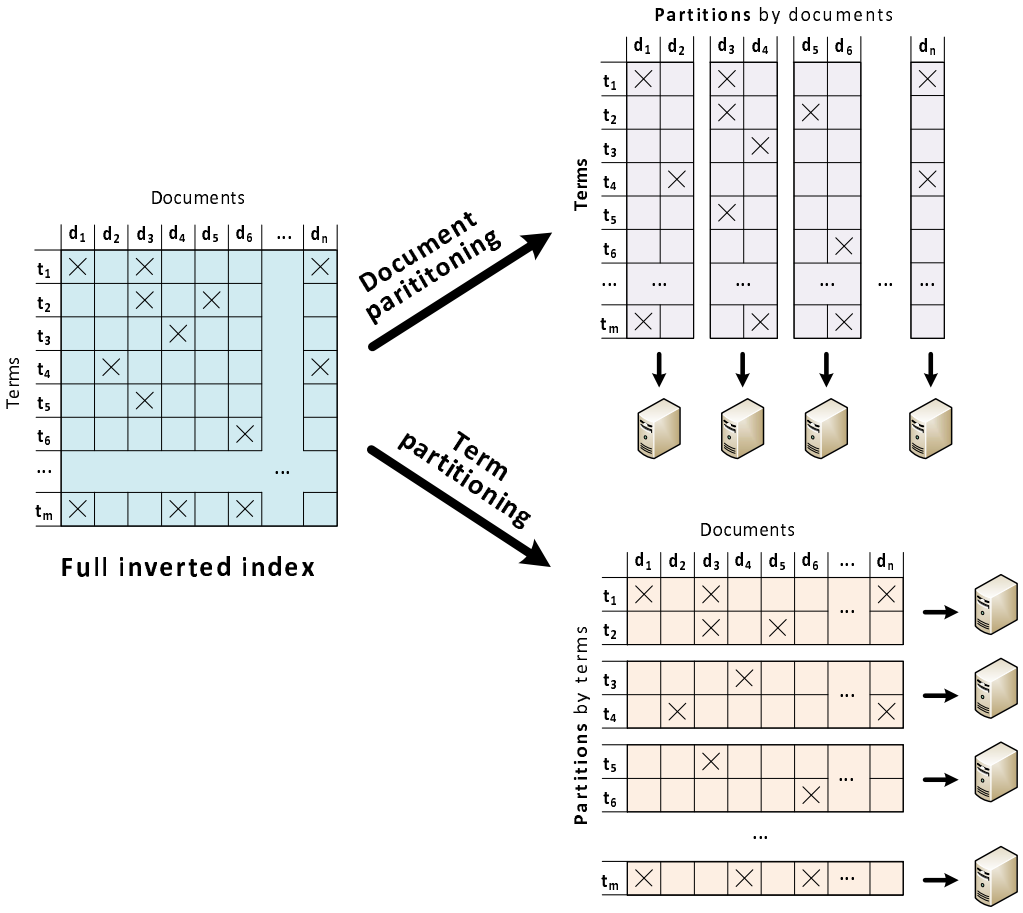

Figure 2.1. Term and document partitioning of the inverted index.

Recent work by [Bender *et al.* 2007] surveys the design alternatives for large-scale Web search focusing on index partitioning in search engines and P2P networks.

[4] We provide evidence of this observation in Chapter 6 by analyzing a real Web search engine query log.

2.2.2. Web Search Engines

In this section we discuss the main architectural principles of Web search engines (WSEs) with an emphasis on caching and index pruning techniques as being relevant to Chapter 6.

Figure 2.2 shows a simplified architecture of a Web search engine. Following this architecture, users submit queries through a front-end server. Upon receiving a new query, such a server forwards it to the back-end servers for processing. Following the document partitioning principle described above, each of the back-end servers maintains an index for a subset of the document collection. Each back-end server resolves the query against its local index. The index comprises posting lists for all terms in the sub-collection, where each posting list contains *(document reference, term frequency)* pairs. Once the servers finish the query processing, they return their results to the front-end server that displays them to the user. A broker machine is usually responsible for aggregating the results from a number of back-end servers, and returning these results to the front-end server. The whole main index can be replicated to handle higher query loads.

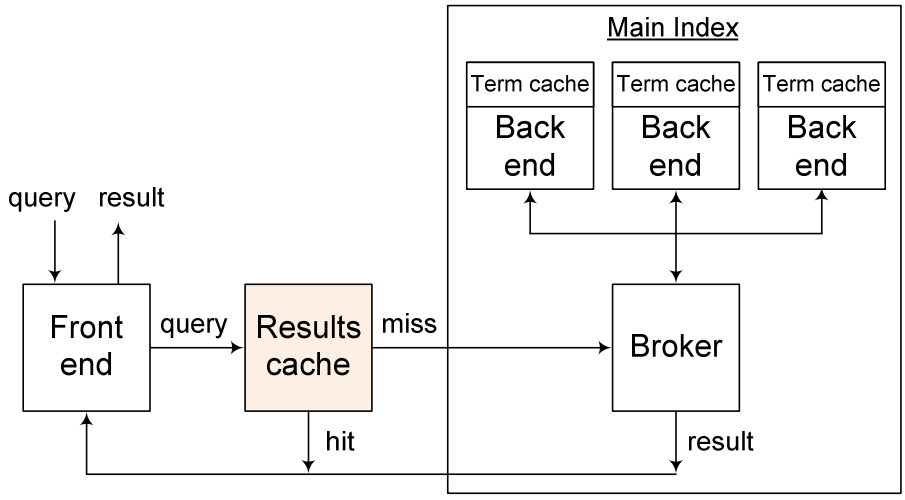

Figure 2.2. A simplified Web search engine architecture.

A results cache is placed in between the front-end and the broker. It maintains a fixed-capacity temporary storage of previously computed top-k query results as the broker has to send them to the front-end server in any case during query processing. Because of the very skewed popularity distribution of Web search queries, results caching can reduce the number of queries that hit the back-end servers by more than 50% [Baeza-Yates *et al.* 2007b]. Each back-end server is also equipped with a local term cache – an in-memory structure that speeds up local query processing by minimizing the number of disk input/output operations.

Given the tremendous impact of search engines on the Internet today a number of solutions that aim at reducing query processing costs in such architectures have been proposed

in the literature, including various caching and index pruning techniques. Furthermore, alternative architectures were proposed, *e.g.*, recent work by [Moffat *et al.* 2007] investigates term-partitioned index strategies for Web search engines. Other approaches that deal with term-partitioning by [Lucchese *et al.* 2007; Zhang and Suel 2007] analyze query-logs in order to efficiently assign terms to index partitions and avoid expensive distributed intersections by placing frequently co-queried terms on the same server.

2.2.2.1. Caching in WSE

Eviction policies that maximize the hit rate for a given cache capacity have been studied in a number of different domains, including search engine architectures. The standard Least Recently Used (LRU) eviction policy discards the least recently used items first. Despite various improvements over LRU were suggested, it is used in many domains due to its simplicity. In the context of **results caching**, [Fagni *et al.* 2006] introduce a Static Dynamic Cache (SDC). SDC achieves higher cache-hit rates than the baseline LRU by devoting a fraction of the storage for a static cache containing frequent queries pre-computed in advance. Further improvements are considered by [Baeza-Yates *et al.* 2007c]. However, when the cache capacity increases, the hit rate approaches its upper bound determined by the fraction of unique queries in the query log. Thus, sophisticated eviction policies have little effect on the performance of a results cache with the capacity of several gigabytes.

[Teevan *et al.* 2007] explore users' repeat searching behavior and categorize different reformulations of similar queries in the query logs. [Baeza-Yates *et al.* 2007b] investigate the impact of results caching and static caching of posting lists in the context of Web search engine architecture. The impact of compression on caching efficiency is addressed by [Zhang *et al.* 2008]. Finally, [Long and Suel 2005] introduce a 3-level caching architecture that includes on-disk caching of the posting lists for popular term combinations.

2.2.2.2. Index Pruning in WSE

Index pruning can significantly reduce the fraction of the index needed for query processing. With *static pruning*, the system generates a pruned index beforehand, whereas *dynamic pruning* proceeds on a per-query basis saving resources and reducing latency by dynamically skipping non-relevant parts of the index.

Index pruning can also be classified as term pruning and document pruning. Term pruning is the complete removal of posting lists of certain terms (*e.g.*, stop words removal or the approach described by [Blanco and Barreiro 2007]), whereas document pruning refers to ignoring only certain portions of the posting lists (*e.g.*, [Carmel *et al.* 2001]).

In particular, some document pruning techniques were inspired by the algorithms introduced by [Fagin *et al.* 2001], known as *threshold algorithms*. The intuition behind these algorithms is that there is no need to scan complete inverted lists if only the top-k fraction of the

intersection is requested. Instead, the lists are sorted according to some score-dependent value and it is likely that the top-k query results can be found in the top portions of the posting lists.

In the context of text search engines this idea was first exploited by [Carmel *et al.* 2001]. They introduced a static document pruning mechanism that is able to reduce the size of the index by up to 50-70% but at the price of a certain loss in precision. [de Moura *et al.* 2005] extend Carmel's lossy pruning by taking into account co-occurrences of words that appear close to each other in documents. A similar pruning technique is also employed by [Long and Suel 2003].

Impact-ordered inverted lists and a lossy dynamic pruning scheme tailored for such an index organization are proposed by [Anh and Moffat 2006]. An impact is a compression-friendly representation of the importance of a term in a document. [Tsegay *et al.* 2007] extend this approach by considering in-memory caching of pruned posting lists.

Alternatively, lossy index pruning based on removal of collection-specific stop words is discussed by [Blanco and Barreiro 2007]. [Büttcher and Clarke 2005, 2006] use a compact pruned index that can fit in the main memory of back-end servers. While [Büttcher and Clarke 2005] combine term and document pruning, the approach described by [Büttcher and Clarke 2006] advocates pruning the least important terms for each document individually. Query processing with the full index maintained in main memory is discussed by [Strohman and Croft 2007].

The approach presented by [Ntoulas and Cho 2007] investigates static index pruning with the aim of reducing the amount of resources needed to handle a given query-rate without sacrificing the result quality (lossless pruning). It employs term pruning, document pruning, and a combination of both. For a real query log and a large document collection the authors report relatively good hit rates (60-70%) achieved with the pruned index of 10-20% of the original index size.

ResIn, described in Chapter 6, studies the impact of results caching on the efficiency of static index pruning. We show that the index pruning performance is fundamentally affected by the changes in the query traffic that the results cache induces. We experiment with real query logs and a large document collection, and show that the combination of both techniques enables an efficient reduction of query processing costs and thus is practical to use in Web search engines. We also observed that investigating the behavior of components of a Web search engine in isolation can sometimes give misleading results without considering the impact they have on each other.

2.2.3. P2P Information Retrieval

In contrast to Web search engines architectures, Peer-to-Peer networks impose serious restrictions on the design of P2P-IR systems, such as:

- Decentralization and lack of control: no centralized components such as brokers or caching servers are available,

- Dynamicity: peers can join and leave at any time – the churn rate is substantially higher than the failure rate among back-end servers,
- Heterogeneity: performance, connectivity and storage capacity of peers might largely vary,
- Limited connectivity: unlike in a stable and fast cluster network, overlay connections often exhibit limited bandwidth and long latencies.

Nevertheless, information retrieval in Peer-to-Peer networks gained significant attention in recent years. Apart from the motivations listed in Section 1.1.3, leveraging a P2P network to deploy a highly distributed search engine can be interesting for several reasons:

- A P2P search engine is a straightforward application that can be built on top of P2P networks,
- A (text) search engine is often an important supplementary functionality to other P2P applications such as peer data management systems (PDMS),
- The lack of cental control and coordination can be beneficial for more objective information search,
- Indexing solutions developed for the P2P setting can be considered in other (*e.g.*, centralized) scenarios.

Peer-to-Peer information retrieval and in particular full text retrieval has been investigated for various P2P network organizations. Search techniques in unstructured networks are usually based on broadcasts, thus suffering from high bandwidth consumption. Hence, approaches based on random walks, content-based routing indexes and hierarchical network solutions have been proposed to reduce the generated traffic in a P2P network.

Many P2P-IR approaches employ *peer-level* document collection descriptions to identify the candidate nodes that can process the query. These descriptions guide the peer-selection process followed by document-level retrieval from the selected peers. Such solutions depend on the quality of the peer-level descriptors and in general perform well for clustered content, when a small subset of peers holds documents relevant to a given query. However, the main problem of such solutions is the increase of the number of generated messages during retrieval with the size of the network in order to maintain acceptable retrieval quality. In contrast to such peer-level solutions, *document-level* indexing approaches do not compromise the retrieval quality when the network size grows but require higher index maintenance costs. Such approaches typically distribute the complete index in a structured P2P network using the term partitioning scheme.

Structured networks are more expensive in terms of maintenance, but offer considerably better search efficiency compared to unstructured networks. For example, a DHT that maintains a term-partitioned global index provides a straightforward P2P-IR solution resolving

single-term lookups by hashing terms into keys. Assuming that the distributed index stores posting lists for all terms found in a document collection, then a multi-term query can be resolved by intersecting these lists for all terms in the query. However, this approach faces significant scalability problems caused by the high traffic costs required for intersecting large posting lists. Thus, a number of solutions have been suggested to resolve this issue.

Popular P2P-IR systems are described in the following section.

2.2.3.1. P2P-IR Systems

Some P2P approaches use resource selection algorithms such as CORI [Callan 2000] or the Kullback-Leibler divergence-based algorithm [Xu and Croft 1999] that aim at selecting a small set of resources that contain a large number of documents relevant to the query. Resources are ranked by their likelihood to return relevant documents and top-ranked resources are selected.

For example, the federated search system described by [Lu 2007; Lu and Callan 2003, 2005] uses a hierarchical P2P network organization. The P2P network consists of hubs that are connected to their leaf nodes according to a clustering algorithm. A query is submitted to one or more initially selected hub nodes. The hub uses its resource selection algorithm to relay the query to its leaf nodes as well as to other neighboring hubs based on their descriptions. A TTL counter is attached to each query to limit the resource utilization. Leaf nodes resolve the query against their local document collections and the results are propagated back to the query originator. Hubs execute a result merging algorithm to aggregate answers from different leaf nodes. [Lu and Callan 2006] improve resource selection by modeling past user behavior to direct the search into the adequate part of the network.

A similar solution is employed by [Balke *et al.* 2005a] with an emphasis on federated digital libraries. Here a super-peer backbone network maintains the information about good candidates for answering a query based on past queries. The approach proposed by [Papapetrou *et al.* 2007] clusters peers into communities, each of the communities having a representative super-peer. All occurrences of a term in a community are published in a backbone DHT built by the corresponding super-peer.

SemreX [Jin and Chen 2008] is a system for sharing desktop literature documents in an unstructured P2P network based on clustering semantically similar peers. Peers are connected based on their content with the addition of long-range links to enable efficient content-based routing.

PlanetP [Cuenca-Acuna *et al.* 2003] gossips Bloom filter based summaries about the peers' document collections to offer content addressable publish/subscribe service in an unstructured P2P network. Inverted peer frequency statistics are used to estimate which peers are good candidates to process a given query.

Minerva [Bender *et al.* 2005a,b] maintains a global index with peer selection statistics in a structured overlay to facilitate the peer selection process. The global index only holds compact, aggregated meta-information about the peers' local indexes to the extent that the

individual peers are willing to disclose. A query initiator selects a few most promising peers based on their published per-term metadata. Subsequently, it forwards the complete query to the selected peers which execute the query locally.

Minerva∞ [Michel *et al.* 2005b] is a P2P-IR system that is based on an order preserving DHT. It relies on Term Index Networks (TINs) storing the global inverted list of a term on several peers. The query is processed by a parallel top-k algorithm involving nodes within TINs and across TINs.

Additionally, the importance of the term co-occurrence statistics has been recognized by [Michel *et al.* 2006]. In this approach term co-occurrences facilitate identifying promising peer-level index entries associated with term combinations. The authors show that such a technique largely improves the peer selection process and consequently the retrieval performance. The usage of keyword correlation statistics for publish/subscribe scenarios is discussed by [Zimmer *et al.* 2008].

In contrast to peer-level solutions, document-level indexing has mainly been applied in structured P2P networks. Since large posting lists are the major concern for such solutions, both [Reynolds and Vahdat 2003] and [Suel *et al.* 2003] have proposed top-k posting list joins, Bloom filters [Bloom 1970], and caching as promising techniques to reduce search costs for multi-term queries. A recent work by [Chen *et al.* 2008] reports 73% traffic reduction by applying optimal Bloom filter settings for DHT-based full text retrieval. However, a study by [Zhang and Suel 2005] shows that single-term indexing is practically unscalable for Web sizes even when sophisticated protocols are combined to reduce retrieval costs.

Top-k query processing inspired by the algorithms of [Fagin *et al.* 2001] has been employed by many P2P-IR approaches to combat the problem of extensive bandwidth consumption. The main idea is to terminate the processing of a query as early as possible and at the same time guarantee (or provide probabilistic guarantees) that the top-k results obtained so far are correct. While resolving a multi-term query using the inverted index there is no need to scan complete posting lists if only a top-k fraction of the intersection is requested. Instead, the lists can be sorted according to the score values and it is likely that the top-k query results can be found by probing the documents found in the top-portions of the posting lists only. Early termination is particulary beneficial for distributed posting list intersections since it has an immediate effect of reducing bandwidth consumption. Top-k query processing algorithms tailored for P2P networks include the Distributed Pruning Protocol (DPP) by [Suel *et al.* 2003], the Three-Phase Uniform Threshold (TPUT) algorithm by [Cao and Wang 2004] and a family of distributed threshold algorithms (DTA) with Bloom filter optimizations by [Zhang and Suel 2005].

A family of approximate top-k query processing algorithms called KLEE are proposed by [Michel 2007; Michel *et al.* 2005a]. With small penalties on the top-k result quality KLEE algorithms significantly reduce the bandwidth consumption while query processing. Each peer maintains a histogram that encodes the distribution of scores in its index. Each cell in the

histogram stores a synopsis: a Bloom filter based structure that represents the set of documents whose scores fall in this cell. This data is used in a 4-steps approximate query execution algorithm, which according to the experiments consumes up to an order of magnitude less traffic than TPUT while preserving the recall at around 80-90% depending on the test collection.

The pSearch system [Tang *et al.* 2004] proposes another approach that places documents onto a DHT network according to their semantic vectors produced by Latent Semantic Indexing (LSI) in order to reduce document dimensionality and guarantee solution scalability. However, as semantic vectors have to be defined a priori, the method cannot efficiently handle dynamic scenarios and adapt to changing collections.

A costly distributed posting list intersection can also be avoided if posting lists, apart from document identifiers, also store so-called document digests – lists of terms contained in the documents. Such document digests are sufficient for local query answering: a query can be resolved from a posting list for any of its terms. Although this insures that the traffic caused by query processing is low, the size of the index becomes very large. Moreover, populating such an index generates substantial traffic. Therefore, this approach is typically used by smaller scale applications in domains other from P2P-IR, for example [Bharambe *et al.* 2004; Cai *et al.* 2004; Tryfonopoulos *et al.* 2005]. The paper by [Tang and Dwarkadas 2004] describes eSearch – a P2P full text retrieval system that uses a similar indexing principle. eSearch stores only selected terms from documents in the posting lists thus sacrificing the search quality in order to reduce the index size. A similar technique is employed by [Li *et al.* 2005] and [Kurasawa *et al.* 2007].

Our Distributed Cache Table (DCT) approach described in Chapter 4 also uses document digests for indexing. In contrast to the approaches described above, it generates the index (or distributed cache) "on-the-fly", driven by the query load, and caches results for multi-term queries instead of single terms only.

The idea of indexing with term combinations has also been exploited in the past. An approach that is close to DCT with respect to the caching strategy is suggested by [Bhattacharjee *et al.* 2003]. The authors employ a similar idea with a different hierarchical index organization called ViewTree. However, no load balancing is considered and no explicit results showing query processing costs are reported. Finally, a recent work by [Kobatake *et al.* 2008] suggests a caching scheme for conjunctive queries in a DHT network that closely resembles the DCT approach.

The Keyword-Set Search System (KSS) [Gnawali 2002] precomputes and stores results of inverted list intersections for popular keywords. However, KSS' exhaustive generation of term combinations leads to unrealistic storage requirements for the index. An extension of the KSS system that takes term co-occurrence statistics into account in order to reduce the index size is presented by [Zhang *et al.* 2005]. However, neither the result quality nor the cost of indexing is reported. Moreover, the method requires a query log available in advance and might not adapt well to future queries.

The HDK approach discussed in Section 2.2.3.2 as well as the indexing scheme presented in Chapter 5 rely on multi-term indexing but also focus on top-k query processing and employ pruning of posting lists to reduce the bandwidth consumption.

A query-driven indexing method at document granularity has recently been proposed by [Li *et al.* 2007]. The solution is based on single-term indexing and, contrary to our contributions, does not consider indexing with term combinations.

The paper by [Loo *et al.* 2004] suggests to avoid maintaining large posting lists in the global index by complementing index-based query processing with broadcasting. The authors suggest using flooding mechanisms to answer popular queries, and resort to indexing only for rare queries. Interestingly, this approach is the opposite to caching-based P2P systems such as DCT.

[Shi *et al.* 2004] suggest a hybrid index partitioning scheme for keyword search. All peers are clustered in groups and the indexing technique employs term partitioning within the groups, but uses document partitioning between the groups. Thus, each query has to be broadcast to all the groups but only several nodes do the actual processing within each group. Since the document collection size within a group can be bounded, this solution reduces latency and efficiently distributes the bandwidth consumption compared to the standard P2P global index approach.

While most of the approaches employ either peer-level or document-level indexing granularity, the recent work by [Nguyen *et al.* 2008] suggests an adaptive scheme aiming at balancing the costs between indexing and query processing. For an individual peer, groups of local documents are created and represented as term sets which are managed by the index. Thus, such a group-level indexing strategy is a generalization of both indexing techniques: peer-level (one group per peer) and document-level (one document per group). The authors propose a probabilistic model to estimate the cost associated with a given number of groups. They also introduce a grouping algorithm based on the observed term distributions derived from the documents and the query logs. Experimental results report cost reductions of $47 - 73\%$ compared to the standard peer-level and document-level indexing strategies respectively.

The approach described by [Joung *et al.* 2005] follows quite a different indexing technique for P2P keyword search. Each document is mapped to an r-bit vector according to its keyword set and is viewed as a point in a r-dimensional hypercube. The hypercube is matched into a DHT such that each document is indexed by one peer only. Query expansion and caching are used to reduce bandwidth consumption for queries containing few keywords.

Finally, [Yong Yang and Cooper 2006] compare the performance of full text retrieval in structured, hierarchical and unstructured P2P systems for information retrieval. They conclude that in their experimental setting random walks in the unstructured network suffer from very high latencies, while the structured network provides the best response times for query processing. On the other hand, the super-peer network consumes less bandwidth than the structured P2P network during indexing and the unstructured P2P network obviously does not require indexing at all. These results, however, depend on the experimental setup and the implementation of each approach.

2.2.3.2. Indexing with Highly Discriminative Keys

In this chapter we present the HDK approach [Luu 2007; Podnar *et al.* 2006b, 2007], a DHT-based full text retrieval solution based on indexing with Highly Discriminative Keys (HDKs). We pay a special attention to this indexing method because we use it as the basis for the query-driven indexing approach in Chapter 5.

Instead of indexing with single terms, which leads to potentially very large posting lists, the HDK approach reduces the retrieval traffic by systematically truncating large posting lists to a constant size, while compensating the resulting loss of information by also indexing with carefully selected combinations of indexing terms. Consequently, the index contains a larger number of entries – terms *and* term combinations which are denoted as *indexing keys*. However, each entry is associated with a shorter posting list.

The idea behind the HDK approach is quite intuitive and is depicted in Figure 2.3. This approach is fully in line with the general properties of P2P networks that can easily store large amounts of data (provided that enough peers are available), but must be carefully controlled with respect to the number of messages and the volume of information transmitted between the peers.

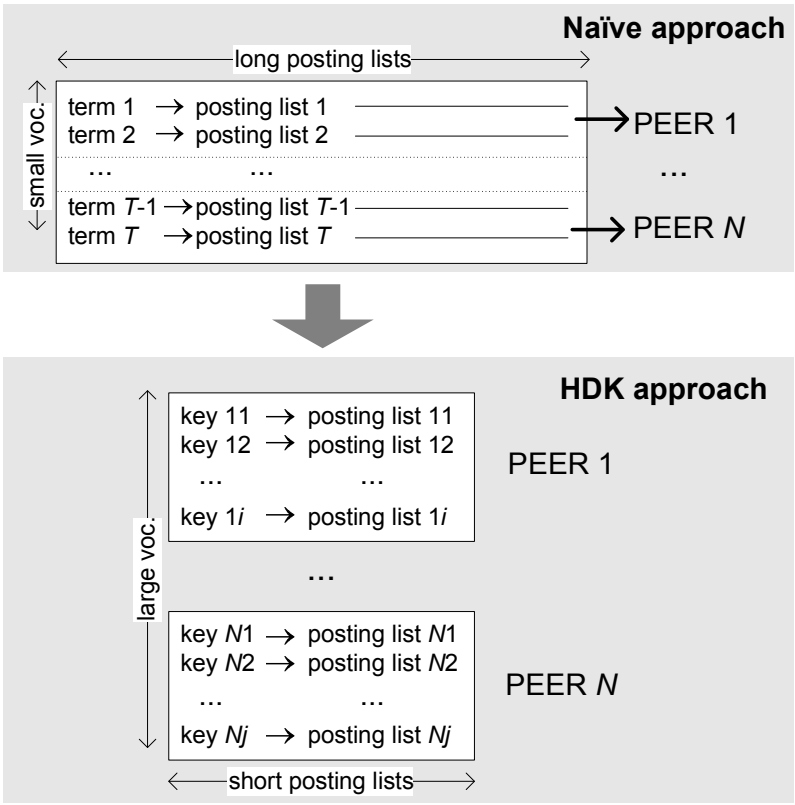

Figure 2.3. The basic idea of HDK indexing: to index a *large* number of *term combinations* associated with *small posting lists* instead of a *small* number of *terms* with *large posting lists*.

Query processing with such an index is performed by collecting the truncated posting lists for the keys that are contained in the query and computing the top-k best ranked results. Bandwidth consumption and latencies are minimal because the size of any posting list is bounded by a constant denoted as DF_{max} (*i.e.*, maximal document frequency). Chapter 5 describes the query processing in more detail.

However, the number of potential multi-term keys grows exponentially with the size of the collection, therefore indexing keys have to be carefully selected such that the index remains scalable, but, at the same time, delivers acceptable retrieval quality. The HDK approach defines a number of filtering conditions for selecting indexing keys, which we briefly describe below.

Size filtering limits the number of terms in an indexing key (or the size of the key) to a maximal size s_{max}. Obviously, setting $s_{max} = \infty$ disables the size filtering and permits keys with an arbitrary amount of terms to be indexed, which is also supported.

Proximity filtering limits the number of documents matching a multi-term key by introducing a proximity window w. A document matches a multi-term key only if all terms from the key are encountered within the distance of w terms from each other within the document. A set of keys K_w denotes all possible keys that pass the proximity filter for a given document collection.

Redundancy filtering relies on the key subsumption property, which is described below.

Each key k is associated with its document frequency $df(k)$ corresponding to the number of documents in the collection that contain k. Given a document frequency threshold DF_{max}, the key document frequencies are used to classify the keys into two distinct categories: *discriminative* keys and *non-discriminative* keys. Discriminative keys (DKs) are the keys that appear in at most DF_{max} documents and therefore have a high discriminative power. On the other hand, non-discriminative keys (NDKs) are the keys with a low discriminative power.

Notice that the DKs (resp. NDKs) verify the following *subsumption property*: any key containing a DK of smaller size is also a DK. Respectively, any key contained in an NDK of bigger size is also an NDK.

The redundancy filtering method relies on the subsumption property of the DKs to further reduce the number of candidate keys. If a key k_1 contains a discriminative key k_2 of a smaller size, then k_1 is also discriminative and the answer set $PL(k_1)$, which is contained in $PL(k_2)$, can be produced by local postprocessing of $PL(k_2)$. In other words, k_1 is practically redundant with respect to k_2 and therefore does not need to be stored in the global index.

Thus, a key k is *highly discriminative* iff: 1) $|k| \leq s_{max}$ (size filtering), 2) $k \in K_w$ (proximity filtering), and 3) k is discriminative and all its sub-keys of strictly smaller size are non-discriminative.

In other words, redundancy-based filtering implies considering only highly-discriminative keys and all their (non-discriminative) sub-keys for indexing. It greatly reduces the number of candidate keys, and, due to the subsumption property, fully preserves the indexing exhaustiveness.

The scalability analysis of the HDK approach [Podnar *et al.* 2007] has shown that the number of generated keys grows linearly with the number of documents, which is acceptable under the reasonable assumption that the ratio between the total number of documents and the total number of peers in the network remains bounded.

However, we have observed that the HDK approach generates a large number of keys that are never or rarely used in queries. Indeed, as the keys are generated only on the basis of their document frequencies, their popularity in user queries (and thus practical usefulness) is not taken into account. Obviously, the creation and maintenance of such superfluous keys causes substantial consumption of both bandwidth and storage, which represent valuable resources in large-scale networks. We describe our solution to this problem in Chapter 5.

Chapter 3

Efficient Processing of XPath Queries in a Peer-to-Peer XML Storage*

3.1. Introduction

This chapter presents an approach that aims at efficient processing of XPath queries over a large XML repository distributed in a structured P2P network. We suggest an indexing structure that is optimized for XPath query processing and extensively employs *caching* of partial results to efficiently handle some XPath constructs. Because of the dedicated caching mechanism, the content of the index is directly influenced by the query load and thus we refer to such an indexing strategy as *query-driven*. This chapter describes the initial results of applying query-driven indexing for P2P search. In Chapters 4 and 5 we will explore this concept in greater detail in the context of the P2P text retrieval scenario.

We assume a structured P2P system that can process queries expressed in a complex XML query language such as XQuery. XQuery uses XPath expressions to locate data fragments by navigating structure trees of XML documents stored in the network. We refer to this functionality as processing of *path queries*. In this work we do not address query plans or joins, but focus on a basic indexing strategy that facilitates efficient answering of path queries, which we refer to as *structural indexing* in the following. We restrict the supported queries to a subset of the XPath language including node tests, child axes ("/"), descendant axes ("//") and wildcards ("*"), which we will denote as $XPath_{\{*,//\}}$. Our goal is to provide a basic functional building block that can be exploited by a higher-level query engine to efficiently answer structural parts of complex queries in large-scale structured P2P systems. Moreover, we think that the work presented in this chapter provides concepts, which can be generalized

* The material presented of this chapter was published in the proceedings of the International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE'05) [Skobeltsyn *et al.* 2005].

to a more complete support of XPath predicates and joins in the P2P environment.

The remainder of the chapter is organized as follows. Section 3.2 gives a brief introduction of the P-Grid structured overlay network, which we use to evaluate our approach. Our basic indexing strategy is described in Section 3.3. In Section 3.4 we present a dedicated distributed caching technique that improves the performance of the basic index. The complete approach is then evaluated in Section 3.5 through simulations. Finally, we present our conclusions in Section 3.6.

3.2. P-Grid

We have briefly introduced P-Grid[1] [Aberer 2001] in Section 2.1 already. However, since its properties are crucial for our indexing strategy and also to keep the chapter coherent we give a more detailed P-Grid overview in the following.

P-Grid is a structured overlay network that implements a Distributed Hash Table (DHT). P-Grid peers refer to a common underlying trie structure in order to organize their routing tables as opposed to other topologies, such as rings (Chord [Stoica *et al.* 2001]), multi-dimensional spaces (CAN [Ratnasamy *et al.* 2001]), or hypercubes (HyperCuP [Schlosser *et al.*]). A trie is a generalization of a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes. In the following we will use the terms trie and tree conterminously.

Without constraining general applicability we use binary keys in P-Grid. Each peer $\pi \in \Pi$, $|\Pi| = N$ is associated with a leaf of the binary tree (see Figure 3.1). Each leaf corresponds to a binary string γ, also called the *key space partition*. Thus, each peer π is associated with a path $\gamma(\pi)$. To construct the routing table, the peer π stores for each prefix $\gamma(\pi, \ell)$ of $\gamma(\pi)$ of length ℓ a set of references $\rho(\pi, \ell)$ to peers π' with the property $\overline{\gamma(\pi, \ell)} = \gamma(\pi', \ell)$, where $\overline{\gamma}$ is the binary string γ with the last bit inverted. Therefore, at each level of the tree the peer has references to some other peers that do not pertain to the peer's subtree at that level. This enables the implementation of prefix routing for efficient search. The cost for storing the references and the associated maintenance cost scale as they are bounded by the depth of the underlying binary tree.

Each peer stores a set of data items $\delta(\pi)$. For the data item $d \in \delta(\pi)$, the binary key $key(d)$ is calculated using an *order-preserving hash function*, *i.e.*, $\forall s_1, s_2 : s_1 < s_2 \Rightarrow h(s_1) < h(s_2)$, which is a prerequisite for efficient range querying as information is being clustered. The binary key $key(d)$ has $\gamma(\pi)$ as prefix, but it is not excluded that temporarily also other data items are stored at the peer. That is, the set $\delta(\pi, \gamma(\pi))$ of data items whose key matches $\gamma(\pi)$ can be a proper subset of $\delta(\pi)$. Moreover, for fault-tolerance, query load-balancing and hot-spot handling, multiple peers are associated with the same key-space partition (structural replication). Additionally, peers maintain references to the peers with the same path, *i.e.*,

[1] P-Grid project web site: `http://www.p-grid.org`

their replicas, and use epidemic algorithms to maintain replica consistency. Figure 3.1 shows a simple example of a P-Grid tree. Notice that while the network uses a tree/trie abstraction, the system is in fact hierarchy-less, and all peers reside at the leaf nodes. This avoids hot-spots and single points of failure.

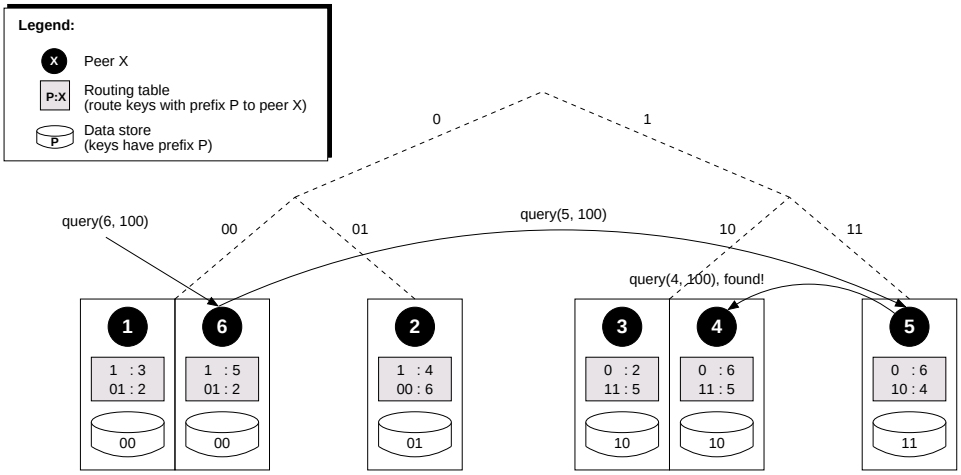

Figure 3.1. P-Grid overlay network.

P-Grid supports a set of basic operations:

- *Retrieve(key)* for searching a certain key and retrieving the associated data item,
- *Insert(key, value)* for storing new data items,
- *Update(key, value)* for updating a data item, and
- *Delete(key)* for deleting a data item.

Since P-Grid uses a binary tree, *Retrieve(key)* is of complexity $O(\log N)$, measured in overlay messages, required for resolving a search request in a balanced tree. *I.e.*, all paths associated with peers are of equal length. Skewed data distributions may imbalance the tree, so that it may seem that search cost becomes non-logarithmic in the number of messages. However, in [Aberer 2002] it is shown that due to the randomized choice of routing references from the complimentary subtree, the expected search cost remains logarithmic, independently of how the P-Grid is structured. The intuition why this works is that in search operations keys are not resolved bit-wise, but in larger blocks, thus the search costs remain logarithmic in terms of messages. This is important as P-Grid's order-preserving hashing may lead to non-uniform key distributions.

The basic search algorithm is shown in Algorithm 3.1. The peer that currently processes the request is denoted as π in the algorithm. The algorithm always terminates successfully, if the P-Grid is complete (ensured by the construction algorithm) and at least one peer in each

partition is reachable (ensured through redundant routing table entries and replication). Due to the routing table construction, $Retrieve(key, \pi)$ will always find the location of a peer at which the search can continue (use of completeness). With each invocation of $Retrieve(key, \pi)$, the length of the common prefix of $\gamma(\pi)$ and key increases at least by one and therefore the algorithm always terminates.

Algorithm 3.1 Search in P-Grid: $Retrieve(key, \pi)$.

1: **if** $(\gamma(\pi)$ *isPrefixOf* $key)$ **or** $(\gamma(\pi) = key)$ **or** $(key$ *isPrefixOf* $\gamma(\pi))$ **then**
2: return$(d \in \delta(\pi) \mid key(d) = key)$;
3: **else**
4: determine ℓ such that $\gamma(key, \ell) = \overline{\gamma(\pi, \ell)}$;
5: $r =$ randomly selected element from $\rho(\pi, \ell)$;
6: $Retrieve(key, r)$;
7: **end if**

Insert(key, value) and *Delete(key)* are based on P-Grid's more general update functionality [Datta *et al.* 2003]. An insert operation is executed in two logical phases: first an arbitrary peer responsible for the key-space to which the key belongs is located ($Retrieve(key, \pi)$) and then the found peer notifies its replicas about the inserted key using a light-weight hybrid push-and-pull gossiping mechanism. Deletions and updates work alike.

Table 3.1 summarizes the main notations we will use throughout the chapter.

π	Peer
N	Number of peers in the P2P network
$\gamma(\pi)$	Path of peer π
$\gamma(\pi, \ell)$	Prefix of length ℓ of π's path
$\rho(\pi, \ell)$	π's routing table references at level ℓ
$\delta(\pi)$	Data items under responsibility of peer π
$key(d)$	Binary key of a data item d
$h(\text{“}val\text{”})$	Result of applying the hashing function on “val”
q_B	Longest subpath of the query q, see Definition 3.1
q_C	Sorted sequence of subpaths of the query q, see Definition 3.2

Table 3.1. Main notations of Chapter 3.

3.3. Basic Index

The goal of structural indexing is to provide efficient means to find a peer or a set of peers, that store pointers to XML documents or fragments containing the path(s) matching the queried expression. As we target large-scale distributed XML repositories, our goal is to minimize the

messaging costs, measured in overlay hops, required to answer the query. The intuition of our approach is to use standard database techniques for suffix indexing applied to XML path expressions. Instead of symbols, the set of XML element tags is used as the alphabet.

Given an XML path P consisting of m element tags, $P = t_1/t_2/t_3/\ldots/t_m$, we store m data items in the P-Grid network using the following subpaths (suffixes) as application keys:

- $sp_1 = t_1/t_2/\ldots/t_m$
- $sp_2 = t_2/\ldots/t_m$
  $\vdots$
- $sp_m = t_m$

The key of each data item is generated using P-Grid's prefix-preserving hash function: $key_i = h(sp_i)$. Insertion of m data items requires $O(m \log N)$ overlay hops. Each data item stores the original XML path to enable local processing and the URI of the XML source document/fragment. We refer to this index as *basic index*.

For example, for the XML path "*store/book/title*", the following data items (we represent them in a form of {key, data} pairs) will be created:

- $\{h(\text{“}store/book/title\text{”}), (\text{“}store/book/title\text{”},\ URI)\}$
- $\{h(\text{“}book/title\text{”}), (\text{“}store/book/title\text{”},\ URI)\}$
- $\{h(\text{“}title\text{”}), (\text{“}store/book/title\text{”},\ URI)\}$

Any peer in the overlay network can submit an $XPath_{\{*,//\}}$ query. To support wildcards ("*") we consider them as a particular case of descendant axes ("//"). They are converted into "//" and are used only at the local lookup stage as a filtering condition. *I.e.*, our strategy is to preprocess a query replacing all "*" by "//", for example, "$A/*/B$" $\rightarrow$ "$A//B$", answer the transformed query using the distributed index and filter the result set using the original query. Thus, we obtain the intended semantics of wildcards. In this approach we concentrate on general indexing strategy and do not address possible optimizations on this issue.

Definition 3.1. A **longest subpath of a query** q, denoted as q_B, refers to the longest sequence of element tags divided by child axes ("/") only. For example, for the query "$A//C/D//F$", $q_B =$ "C/D".

When a query is submitted to a peer, the peer generates the query message that contains the path expression and the address of the originating peer. This message is sent to the network following the basic structural querying algorithm as shown in Algorithm 3.2. The basic algorithm uses the longest query subpath q_B as a *search key*[2].

[2] We choose the q_B notation for the search key because it is used with the *basic index*.

Algorithm 3.2 XPath querying using the basic index: $AnswerQueryBasic(q, \pi)$.

```
1: compute q_B from q;
2: key = h(q_B)
3: if (γ(π) isPrefixOf key) or (γ(π) = key) then
4:    return(d ∈ δ(π) | isAnswer(d, q) = true);
5: else if key isPrefixOf γ(π) then
6:    ShowerBroadcast(q, length(key), π);
7: else
8:    determine ℓ such that γ(key, ℓ) = γ(π, ℓ)‾;
9:    r = randomly selected element from ρ(π, ℓ);
10:   AnswerQuery(q, r);
11: end if
```

The function $AnswerQueryBasic(q, \pi)$ in Algorithm 3.2 extends the default P-Grid's routine $Retrieve(key, \pi)$ described in Algorithm 3.1 for answering $XPath_{\{*,//\}}$ queries using the basic index. First, the search key key is computed by hashing the query's longest subpath q_B (lines 1–2). Then the algorithm checks whether the currently processing peer is the only one responsible for the key key (line 3). If yes, the routing is finished and the result set is returned (line 4). Function $isAnswer(d, q)$ examines whether the data item d is a correct answer for the query q.

Alternatively, if routing is finished at one of the peers from the sub-trie defined by the key key (line 5), all peers from this sub-trie could store relevant data items and have to be queried. In this situation, the key is the prefix of the peer's path, which means that all bits of the key have been resolved and the query has reached the sub-trie, in which potentially all peers may store the data belonging to the query answer set.

To query all the peers in the sub-trie, we use a variant of the broadcasting algorithm (line 6) for answering range queries described by [Datta *et al.* 2005] as shown in Algorithm 3.3, where the range is defined by the binary prefix key. *I.e.*, we query all the peers for which key is the prefix of their paths.

Algorithm 3.3 Shower broadcast algorithm: $ShowerBroadcast(q, \ell_{current}, \pi)$.

```
1: for ℓ = ℓ_current to length(γ(π)) do
2:    r = randomly selected element from ρ(π, ℓ);
3:    ShowerBroadcast(q, ℓ + 1, r);
4: end for
5: return(d ∈ δ(π) | isAnswer(d, q) = true);
```

The algorithm starts at an arbitrary peer from the sub-trie, and the query is forwarded to the other partitions in the trie using the peer's routing table. The process is recursive, and since the query is split in multiple queries, which appear to trickle down to all the key-space

partitions in the range, we call it *shower algorithm*.

With the basic index the expected cost (in terms of overlay messages) of answering a single query is $L + S - 1$, where L is the cost of locating any peer in the sub-trie and S is the shower algorithm's messaging cost. The expected value of L is a length of the sub-trie's binary prefix. The intuition for this value is that it is analogous to the search cost in a tree-structured overlay of size 2^L. The expected value of S is $N/2^L$, which refers to the number of peers in the sub-trie. The latency remains $O(\log N)$ because the shower algorithm works in a parallel fashion.

To illustrate how a query is answered using the basic index, assume the query "$A//C/D//E$" is submitted at some peer π. Following Algorithm 3.2 the peer responsible for $h($"C/D"$)$ is located. Assume that there is a sub-trie defined by the prefix $h($"C/D"$)$ as it is depicted in Figure 3.2. The shower broadcast is executed and every peer in the sub-trie performs a local lookup for the original query and sends the result to the originating peer π.

3.4. Caching Strategy

The basic index is efficient in finding all documents matching an $XPath_{\{*,//\}}$ query expression based on the longest sequence of element tags q_B. It performs well for the queries containing a relatively long $h(q_B)$, such that the number of peers contacted by the shower broadcast is not excessive. However, the search cost might be substantially higher for queries, which require large shower broadcasts, *i.e.*, $h(q_B)$ is short. For example, queries like "A//B" are answered by looking up the peer responsible for $h($"A"$)$ and then a relatively expensive broadcast depending on the data in the overlay may have to follow. Search would be more efficient if the knowledge about the second element tag "B" was employed as well. In this section we introduce a caching strategy to address this issue, which allows us to reduce the number of broadcasts, and thus, decrease the average cost of query processing.

Each peer upon receiving a query determines whether it belongs to one of the following types:

1. Queries that can be answered locally, *i.e.*, only a single peer is responsible for q_B: $\gamma(\pi)$ is a prefix of $h(q_B)$ or $\gamma(\pi) = h(q_B)$. For example the query "$A/B/C//E$" at the peer responsible for $h($"A/B"$)$.

2. Queries that require an additional broadcast, *i.e.*, $h(q_B)$ is a prefix of $\gamma(\pi)$, but contain only one subpath such that $q = q_B$. For example, the query "A" at the peer responsible for $h($"A/B"$)$. In this case the matching index items are stored on all the peers responsible for $h($"A"$)$. As queries of this type may be very expensive (for example "//"), they could be disabled in the configuration or only return part of the overall answer set to constrain costs.

3. Queries that require an additional broadcast, *i.e.*, $h(q_B)$ is a prefix of $\gamma(\pi)$, but include at least one descendant axis ("//") or wildcard ("*") such that $q_B \neq q$. For example the

query "$A//C//E$" at the peer responsible for $h(\text{"}A/C\text{"})$. The result set for such queries can be cached locally and accessed later without performing the shower broadcast.

Type 1 queries are inexpensive and work well with the basic index. *Type 2 queries* are so general that they return undesirably large result sets and the system may want to block or constrain them. The most relevant type of queries whose costs could be minimized are, thus, *type 3 queries*, which we will address in the following. For simplifying the presentation we assume that only one peer is responsible for a given query and has sufficient resources to cache results.

3.4.1. Answering a Query

In order to enable the caching mechanism we modify the routing process and add cache management to the "basic" Algorithm 3.2.

Definition 3.2. A **sorted sequence of subpaths** of a query q, denoted as q_C, is obtained by sorting all q's subpaths[3] by their length in descending order and concatenating them: $q_C = concat(P_{l_1},\ P_{l_2},\ ...,\ P_{l_k})$, where P_{l_i} is the i-st longest subpath. For example, for the query $q =$"$A//C/D//F$", $q_C =$ "$C/D/A/F$" using "/" as the concatenation symbol.

We will use q_C for routing instead of q_B, which gives us the benefit that we use the whole query for generating the search key[4], thus making it more discriminative. Clearly for type 3 queries $|q_C| > |q_B|$. The modified querying algorithm is shown in Algorithm 3.4.

In line 1 we compute q_C which is used for routing (line 12) to the peer possibly storing the cached result set. Since P-Grid uses a prefix-preserving hash function and $q_B \subseteq q_C$ (q_B is always the first subpath of q_C), this peer is located in the $key_B = h(q_B)$ sub-trie.

Similarly to the basic index's search algorithm we check whether the currently processing peer is the only one responsible for key_B (line 5). If yes, the result set is returned (line 6). If the routing reached one of the peers from the sub-trie defined by the key key_B, we execute the shower broadcast algorithm (line 8) to answer the query as introduced in the previous section, but only if the query has not already been cached (line 7). Section 3.4.2 explains how the function $ifCached(q)$ works. If the query is cached, the routing proceeds until the peer responsible for the key key_C is reached. This peer answers the query by looking up the cached result set (line 10).

[3] Recall, that a subpath of a query is a sequence of element tags divided by child axes ("/") only.

[4] We choose the q_C notation for the search key here because it is used with the *basic index plus caching*, contrary to q_B, which is used with the *basic index* only.

Algorithm 3.4 XPath querying using the basic index extended with caching: $AnswerQueryCachingEnabled(q, \pi)$.

1: compute q_C from q;
2: $key_C = h(q_C)$
3: compute q_B from q;
4: $key_B = h(q_B)$
5: **if** $(\gamma(\pi)\ isPrefixOf\ key_B)$ **or** $(\gamma(\pi) = key_B)$ **then**
6: return$(d \in \delta(\pi) \mid isAnswer(d, q) = true)$;
7: **else if** $(key_B\ isPrefixOf\ (\gamma(\pi)))$ **and** $(ifCached(q) = false)$ **then**
8: $ShowerBroadcast(q, length(key_B), \pi)$;
9: **else if** $(\gamma(\pi)\ isPrefixOf\ key_C)$ **or** $(\gamma(\pi) = key_C)$ **then**
10: return$(d \in cache(\pi) \mid isAnswer(d, q) = true)$;
11: **else**
12: determine ℓ such that $\gamma(key_C, \ell) = \overline{\gamma(\pi, \ell)}$;
13: r = randomly selected element from $\rho(\pi, \ell)$;
14: $AnswerQueryCachingEnabled(q, r)$;
15: **end if**

3.4.2. Cache Maintenance

Each peer runs a cache manager, which is responsible for cache maintenance. The cache manager implements two functions $createCache(q)$ and $deleteCache(q)$, where q is any query the peer is responsible for. In the following we explain how these functions work.

To cache a query, a peer determines the sub-trie's prefix by hashing q_B and collects the result set for the query by executing a special version of the shower broadcast algorithm. The only difference compared to the original $ShowerBroadcast$ algorithm (Algorithm 3.3) is that for cache consistency reasons all the peers in the broadcast sub-trie add the query expression to their *lists of cached queries* L_{CQ}. Thus, in case the P-Grid is updated, *i.e.*, data items are inserted, modified or deleted, any peer from the sub-trie can contact the peer(s) that cache relevant queries, to inform them of the change so they can keep their caches consistent. This operation requires $O(\log N)$ messages per cache entry. The function $ifCached(q)$ (line 7, Algorithm 3.4) looks up the locally maintained L_{CQ} list to determine if the query is cached.

When a data item is inserted, updated or deleted, all relevant cache entries are updated respectively. The peer looks up the cached queries list and sends the update messages to all the peers caching the relevant queries. Each cache update requires a message to be routed with $O(\log N)$ cost.

3.4.3. What to Cache?

The cache manager analyzes the benefits of caching for each candidate query the peer is responsible for. To do so, it estimates the overall messaging cost for the query with and without caching. The decision to cache the query result or to delete the existing cache entries is based on comparing these two values.

If the query is cached, each search operation for the query saves one shower broadcast (the shower broadcast requires $s - 1$ messages where s is the number of peers in the trie). On the other hand, each update operation for any data item related to the query will cost additional messages to update the cache. Knowing the approximate ratio of search/update operations (obtained by local monitoring), the peer can make an adaptive decision on caching of a particular query.

The query is considered to be profitable to cache if:

$$UpdateCost * UpdateRate(subtrie) < SearchCost(subtrie) * SearchRate(query),$$

where:

- *subtrie* is the sub-trie described by the q_B prefix, *i.e.*, the basic index's shower broadcast sub-trie;
- $UpdateCost$ is the cost of one update, which is equal to the routing cost;
- $UpdateRate(subtrie)$ is the average update rate in the given sub-trie;
- $SearchCost(subtrie)$ is the number of peers in the sub-trie to be contacted to answer the shower broadcast; and
- $SearchRate(query)$ is the search rate for the given query.

To estimate these values each peer collects the corresponding statistics. For $SearchRate$ the peer's local knowledge is sufficient, whereas the $UpdateCost$ and $SearchCost$ values have to be gathered from the neighbors. To do so, we can periodically flood the network or employ a more efficient algorithm described in [Albrecht *et al.* 2003]. This algorithm gossips the information about the tree structure among all the peers in the network. Each peer maintains an approximate number of peers in each sub-trie it belongs to (as many values as the peer's prefix length). The values are exchanged via local interactions between peers and a piggyback mechanism avoids sending additional messages. The same idea can be used to gossip the $UpdateRate$ in every sub-trie a peer belongs to.

3.4.4. Example

An example illustrating the application of caching is shown in Figure 3.2.

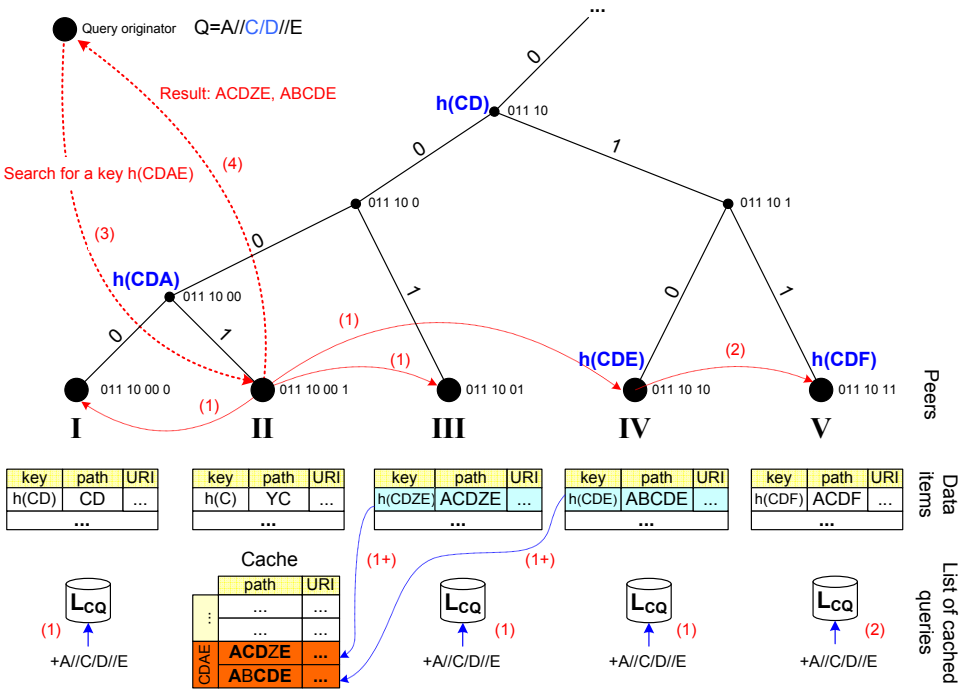

Figure 3.2. Caching strategy example.

Notice that in Figure 3.2 each element tag is represented by one capital letter and we omit child axes ("/") to simplify the presentation. The numbers 1–4 written in brackets next to the arrows correspond to the following steps:

1. The cache manager at the peer II decides to cache the result set for the query $Q =$ "$A//C/D//E$". The shower broadcast to the peers responsible for $h($"C/D"$)$ is initiated to populate the cache with all the data items matching the query. It reaches the peers I, III and IV. They add Q to their lists of cached queries (L_{CQ}).

2. Peers III and IV send back the matching items (not shown). The shower broadcast reaches peer V, which also adds Q to its list of cached queries. Four messages are sent to execute the shower broadcast in the sub-trie $h($"C/D"$)$.

3. Assume, the query $Q =$ "$A//C/D//E$" is submitted at the originating peer again. The search message is routed to the peer II, which can answer the query locally by looking up its cache. The broadcast has to be executed every time to answer the query Q only if it is not cached.

4. The answer is sent back to the originating peer.

Let us now demonstrate how cache updates are handled. Assume a new path "$A/C/D/E$" is indexed and one of the four generated data items[5] with the key $h($"$C/D/E$"$)$ is added to

[5] The query has 4 subpaths, see Section 3.3 for more details.

the peer V. It checks the list of cached queries and finds the cache for Q = "$A//C/D//E$" to be affected by the new path "$A/C/D/E$". Peer V sends the cache update message containing the new path to the Peer II (responsible for the Q's cache) to ensure the cache consistency.

3.5. Simulations

To justify our approach and its efficiency, we implemented a simulator of a distributed XML storage based on the P-Grid overlay network. The simulator is written in Java and stores all data locally in a relational database.

As the input data for our experiments, we use about 50 real XML documents (mainly taken from `http://www.cs.washington.edu/research/xmldatasets`) from which we extracted a *path collection* of more than 1,000 unique paths. Based on each path in the collection we generated four additional paths by randomly distorting the element tags. Using the resulted path collection (about 5,000 paths) we generate a P-Grid network by inserting a corresponding number of data items per each path (about 20,000 data items in total). P-Grid networks of different sizes can be built by limiting the maximum number of data items a peer can store.

For our experiments we generate *queries* by randomly removing some element tags from the paths in the path collection. The parameter t affects the query construction and specifies the fraction of "cachable" (type 3) queries in the collection.

To simulate the querying process we generate several *query logs* of 10,000 $XPath_{\{*,//\}}$ queries each using different distributions.

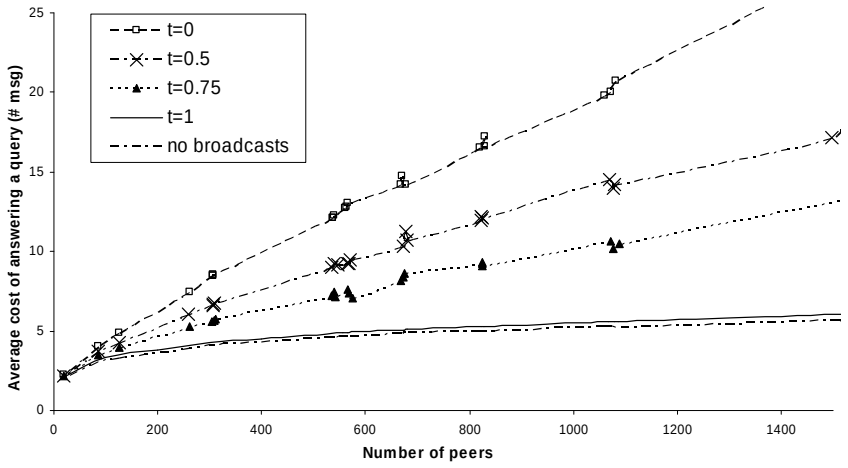

Figure 3.3. Average number of messages required to answer a query depending on the network size, t denotes the fraction of "cacheable" queries.

In the first experiment we assume that all possibly "cacheable" queries are cached. We use different values of the t parameter and the uniform distribution to generate the query logs. We vary the network size and measure the average cost of query processing. In Figure 3.3 the first four curves show the average search cost for t = 0, 0.5, 0.75 and 1 respectively. From

the figure, the more queries are cached, the lower the search cost is. The fifth curve, called "no broadcasts", shows the cost of locating at least one peer responsible for the query, *i.e.*, the search cost when shower broadcasts are ignored. Evidently, the two last curves coincide because if all queries are cached – no shower broadcast is required.

However, in reality query logs do not necessary follow the uniform distribution. Instead, many studies report that real queries are power-law distributed (*e.g.*, in Chapter 6 we will show that this is true for real-word Web search engines query logs). In the next experiment we use the Zipfian distribution to generate the query logs, fix the network size to 1,000 peers, $t = 0.5$ and vary the cache size. The first curve in Figure 3.4 shows the constant search cost when caching is disabled. The other three curves correspond to the different parameters of the Zipf distribution ($s = 0$, 0.8 and 1.2) used to generate the corresponding query log. Unsurprisingly, caching performs better with more skewed query distributions.

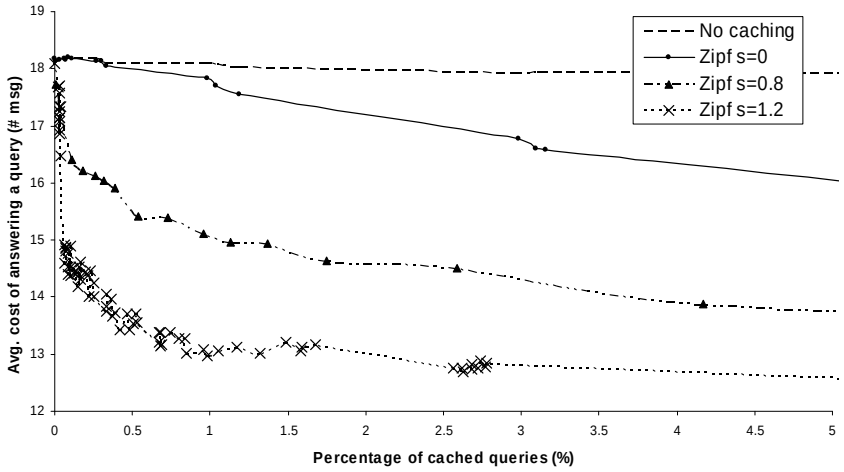

Figure 3.4. Average number of messages required to answer a query in the network of 1000 peers depending on the fraction of cached queries.

However, the benefits we gain from caching at the querying time come at the price of increasing the update costs. To perform one update operation, for example to insert a new path containing m element tags, we have to contact all the peers responsible for all the subpaths (m routing requests). To ensure cache consistency, we also have to update all relevant cache entries (one routing request per cache entry). Figure 3.5 shows the average update costs depending on the size of the network for different numbers of cached queries.

In Section 3.4.3 we described the strategy for minimizing the overall messaging costs. In the last experiment we show that for a given state of the system this minimum can be achieved by choosing which queries to cache. In Figure 3.6 we show that for the given parameters (1,000 peers, $t = 0.5$, Zipf $s = 1.2$, average number of element tags in the path = 2.5) the overall messaging cost can be minimized. We show two curves for search/update ratios of 1:2 and 2:1. In these cases the minimal messaging costs are achieved if about 0.5% and 1.0% of the queries are cached.

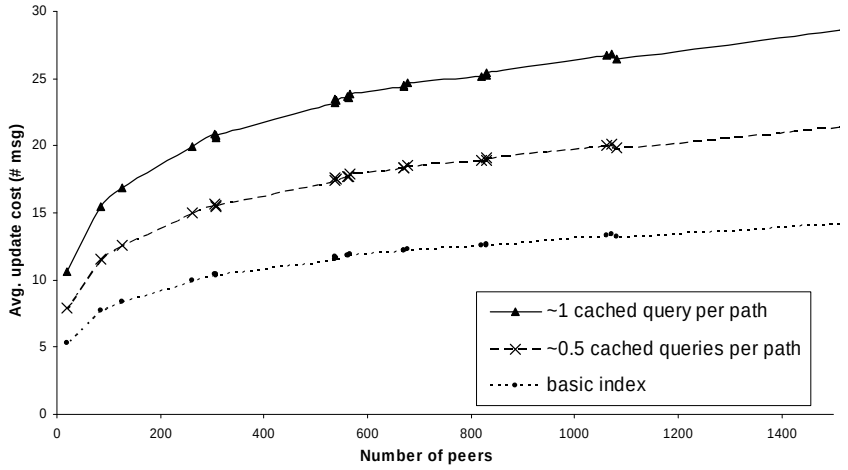

Figure 3.5. Average update cost depending on the network size, t denotes the percentage of "cacheable" queries.

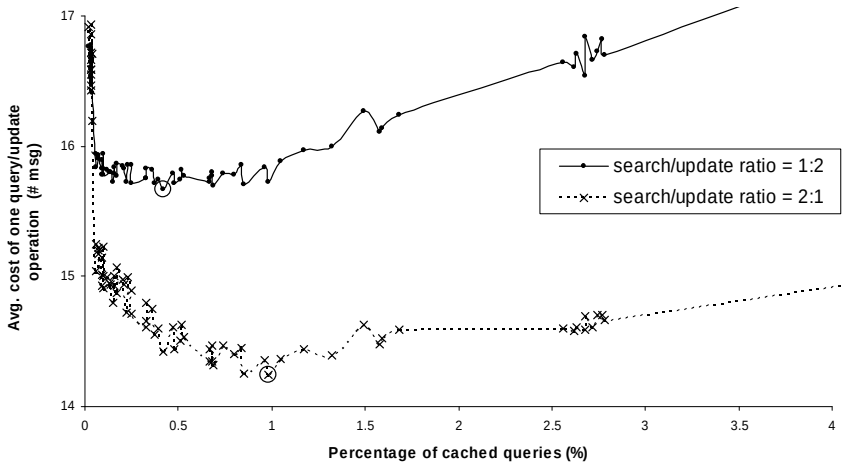

Figure 3.6. Average number of messages (query processing + updates) depending on the fraction of cached queries.

Evidently, if the search/update ratio is high (more searches than updates) the minimum moves to the right (more queries to be cached). In contrast, if the update ratio is relatively high, the minimum moves to the left (up to 0, where caching is not profitable anymore). Hence we observed that our solution can reduce the query processing costs by adapting to the current state of the system.

The simulations show that the basic index strategy is sufficient for building a P2P XML storage with support for processing of path queries. The introduction of caching decreases the messaging costs. Depending on the characteristics of the query load the benefits from caching vary.

3.6. Conclusions

In this chapter we presented the solution for indexing XML data in a structured P2P network. We demonstrated the efficiency (low search latency and low bandwidth consumption) of our approach via simulations. We also showed that our proposed caching strategy reduces messaging costs by adapting to the query patterns and the search/update ratio.

We envision that the presented solution can be used in a P2P XML querying engine for answering structural queries. Such a system could be an alternative to the solutions based on the unstructured P2P networks (*e.g.*, Edutella [Nejdl *et al.* 2002]), but is more scalable due to the considerably reduced messaging costs.

Last but not least, the presented approach is the first evidence that we obtained during the work on this thesis that shows the usefulness of the concept of query-driven indexing in a P2P setting.

Chapter 4

Distributed Cache Table: Query-Driven Indexing for Peer-to-Peer Text Retrieval*

4.1. Introduction

We continue to explore query-driven indexing strategies for distributed query processing in structured P2P networks. In this chapter we look at the P2P text retrieval scenario. We propose an approach for efficient processing of *multi-term* queries over a large collection of textual documents distributed in a P2P (DHT) network. To reduce the overall bandwidth consumption in the network we introduce a query-driven indexing strategy which generates and maintains only those index entries that are actually used for query processing. Such a query-driven indexing structure can be viewed as a distributed cache facility maintained in the P2P network. Thus, by analogy with Distributed Hash Tables (DHTs) we call our approach *Distributed Cache Table* (DCT).

In DCT we suggest to abandon the difference between data indexing and query caching in a structured P2P network, and to store result sets (caches) for the most profitable queries at the nodes of the network. DCT employs a distributed index to efficiently locate caches that can answer a given multi-term query and broadcasts the query to all the peers only if no such caches were found. Evaluations on real Wikipedia data and query logs show that DCT converges to a high cache-hit rate and offers an elegant distributed solution for storing and efficient querying of large amounts of documents in a P2P network. In our experimental setting DCT achieves two orders of magnitude improvement in traffic consumption compared to the standard single-term indexing approach.

As the number of peers in the network can be large, the use of an unstructured network

* The material presented of this chapter was published in the proceedings of the Workshop on Information Retrieval in Peer-to-Peer Networks (P2PIR'06) [Skobeltsyn and Aberer 2006].

is expensive due to high messaging costs induced by frequent broadcasts. Thus, we employ a standard DHT approach to associate queries to their result sets in a non-trivial fashion. Our approach is motivated by the observation that a large number of index entries may never be queried and therefore the maintenance of such entries is unnecessary. Thus, we employ an indexing/caching strategy for efficient processing of multi-term queries that is driven by the *query load*.

DCT populates the storage space provided by participating peers with the results for popular queries and uses this data to answer further queries. Each cached query result contains sufficient information to resolve any other (more discriminative) query, whose answer is contained in the cached result. Peers only maintain caches that are: 1) frequently used to answer queries, and 2) consume little space. DCT performs an adaptive selection of queries to cache, based on the monitored query statistics taking into account limited storage capacity with the goal of minimizing the number of cache-misses. In particular, each peer locally runs a greedy algorithm leading to a global quasi-optimal cache selection.

To get a general idea about our approach consider a scenario where N peers form a DHT-based P2P network. Each peer shares some of its documents with other peers and is able to search documents located at other peers by issuing multi-term queries. A straightforward solution is to broadcast queries to all peers, so each peer can evaluate each query locally. Unfortunately frequent broadcasts generate excessive amounts of redundant traffic.

Let us assume now, that every peer π can provide a limited storage space s_π. The whole network then has $S = \sum_{i=1}^{N} s_{\pi_i}$ storage capacity that peers utilize to store query caches.

Definition 4.1. In DCT, a **cache** for the multi-term query q stores its *result set* RS_q, which contains documents digests (see Definition 4.2) for all documents satisfying q. Document digests enable query filtering, *i.e.*, we can locally answer q using $RS_{q'}$ if $RS_{q'}$ contains RS_q.

Definition 4.2. A **document digest** is a data structure that represents a text document. It contains a unique document identifier and a list of terms extracted from the document.

Now, to answer a multi-keyword query q we first try to find (at least one) cache which can answer q, and issue a broadcast only if no such cache was found. To entirely benefit from caching, DCT exploits *query subsumption*. Hence, we are interested in locating any cache which *contains* the result set of q. The DCT network evolves in time into a distributed cache driven by the query load, avoiding maintenance of (almost) never used indexing information and employing broadcasts only for rare queries.

DCT employs a distributed ranking mechanism described by [Podnar *et al.* 2006a]. When a stored result set (cache) is used to answer a query, it processes the query locally and returns the top-k documents only. The next portion of results can be supplied on demand. This mechanism significantly reduces the traffic consumption during retrieval.

In summary, the main contributions of the DCT approach described in this chapter are the following:

- We introduce a novel query-driven indexing strategy for multi-term queries in a P2P environment that is based on the query subsumption property (see Definition 4.3);
- We perform experiments with real query logs that show a high number of subsumption dependencies in realistic query loads that are beneficial for our approach;
- We achieve a significant overall traffic reduction compared to the distributed single-term indexing approach.

The rest of the chapter is organized as follows. We describe the caching strategy in Section 4.2, in particular, distributed meta-index in Section 4.2.1 followed by the cache management discussion in Section 4.2.2. We discuss load balancing issues in Section 4.3. Simulation results are presented in Section 4.4 followed by the conclusion in Section 4.5.

4.2. Indexing and Caching Strategy

Let us assume a network of N peers, π_i, $i \in 1..N$, where each peer hosts a part of the document collection D_{π_i} and issues queries from a local query load L_{π_i}. Therefore, $D = \bigcup_{i=1}^{N} D_{\pi_i}$ is the global document collection and $L = \bigcup_{i=1}^{N} L_{\pi_i}$ is the global query load.

Let us define a superset $T = \{t_1, t_2, .., t_m\}$ as the vocabulary consisting of all single terms found in the global query load L. Then, a query $q \in L$ is defined as $q = \{t_1, t_2, .., t_n\}$ and q is a subset of T, $q \in 2^T$. The number of terms in q is denoted as $|q| = n$. Similarly, we define a document $d \in D$ as $d = \{t_1, t_2, .., t_r\}$ which is also a subset of T: $d \in 2^T$. Essentially, we simplify the representation of an original document d_0 by intersecting the set of all terms contained in d_0 with T, $d = d_0 \cap T$, and therefore ignore the terms contained in the original document which do not appear in the query load.

A query q matches a document d iff $q \subseteq d$. The result set RS_q for the query q is the set of all documents matching q, $RS_q = \{\forall d_i \in D \mid q \subseteq d_i\}$.

Definition 4.3. We define the **query subsumption** relation as follows. A query q' subsumes a query q when all terms in q' are also contained in q, *i.e.*, $q' \subseteq q$. Obviously, $q' \subseteq q$ implies $RS_{q'} \supseteq RS_q$. Thus, a query q can be answered by postprocessing of the result set associated with q'. For example the query $q' = \{EPFL\ Switzerland\}$ subsumes the query $q = \{EPFL\ Lausanne\ Switzerland\}$.

The set of all possible queries over T can be represented as a *lattice* of the size $2^{|T|} - 1$. Each lattice node corresponds to a query, and the whole lattice models the set of all potential

queries over T that might appear in the query load L. For example, a lattice generated for the vocabulary of four terms $T_{abcd} = \{a, b, c, d\}$ is shown in Figure 4.1.a. An arrow from a query q_1 to a query q_2 reflects the subsumption relation $q_1 \subseteq q_2$. Figure 4.1.b highlights all descendants of nodes a and cd, referring to all the queries that are subsumed by the queries a and cd. Indeed, all queries containing either the term a or both terms c and d can be answered from the two result sets RS_a and RS_{cd}.

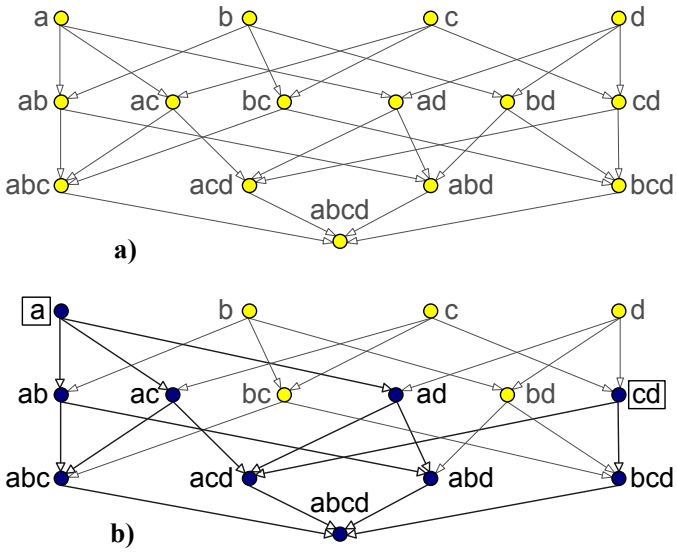

Figure 4.1. Query subsumption: a) a,b,c,d lattice; b) lattice with the queries "a" and "cd" being cached.

Each peer provides a certain cache capacity and uses it to store carefully selected result sets. More precisely, a peer π caches result sets for certain queries from its local query load L_π and advertises them to other peers using a distributed *meta-index*. Therefore, to answer a new query, another peer may lookup the location of existing caches that may resolve the query as it is described in Section 4.2.1. Furthermore, as the peer storage capacity is limited, each peer runs a greedy cache-selection algorithm as described in Section 4.2.2.

Table 4.1 summarizes the main notations used in the chapter.

π	Peer
N	Number of peers in the P2P network
D	Global document collection, D_π is the document collection at the peer π
L	Global query log, L_π is the local query log at the peer π
q	A multi-term query $q = \{t_1, t_2, .., t_n\}$
RS_q	A result set of a query q

Table 4.1. Main notations of Chapter 4.

4.2.1. Meta-Index

The meta-index enables efficient lookup of the result set (cache) locations for a given query. Formally, given q we need to obtain a result set RS_q by locating at least one cache $RS_{q'}$ such that $RS_{q'}$ *contains* RS_q ($RS_{q'} \supseteq RS_q$). In other words, we are interested in locating a cache for q', such that $q' \subseteq q$. To locate relevant result sets, we introduce a distributed meta-index, which stores links to actual cache locations. Given a query q, the meta-index returns a list of tuples $\{q_i,\ uri(RS_{q_i})\}$ for the queries that are: 1) cached, and 2) subsume q. A *random* tuple from the received list is selected and the query q is forwarded to the peer storing the chosen cache. This peer processes q locally and returns the list of documents matching q. If no caches were located by using the meta-index, *i.e.*, the query can not be answered from the cache, it is broadcast to all the peers in the network that evaluate the query against their local document collections and send the answers to the originating peer.

Since peers participate in a DHT, we can use the *shower broadcast* technique [Datta *et al.* 2005] described in Algorithm 3.3. Recall that the shower broadcast insures that each peer is visited only once. To answer a query q, $O(N)$ messages have to be sent to notify all peers that generate $|RS_q|$ records of traffic while answering, where $|RS_q|$ denotes the number of records in the result set of q.

The meta-index is implemented using the standard *put/get* functionality offered by the DHT[1]. Given the cache $RS_{q'}$ physically stored at $uri(RS_{q'})$, an *advertise* operation is performed by inserting a tuple $\{q',\ uri(RS_{q'})\}$ at the peer responsible for the $key = h(t_r)$, where $h()$ denotes the DHT's hash function and $t_r \in q'$ is a *randomly* chosen term from q'. Therefore, the advertise operation requires one *put* message to be send with $O(\log N)$ overlay hops.

Given a query q to be answered, the meta-index lookup operation is performed in the following way: $n = |q|$ messages containing the original query q are sent to the n peers responsible for $h(t_1),\ h(t_2),\ ..,\ h(t_n)$, where $t_i \in q, \forall i \in 1..n$. Each peer responds with a list of cached result set locations for queries that subsume q. Therefore, the cache lookup operation requires n *get* messages to be send with $O(\log N)$ overlay hops each.

Section 4.2.2 introduces the cache management and explains how a peer can locally decide which caches have to be created or evicted leading to a quasi-optimal utilization of the overall network storage capacity S, thus reducing the number of broadcasts for the current query load.

4.2.2. Cache Management

Having defined the meta-index, we can formulate the problem of finding an optimal set of caches in the network, which maximize the cache-hit ratio for: 1) a given query load, and 2) a P2P network with a constrained global storage capacity distributed among the participating peers.

[1] Notice that unlike the XPath indexing approach described in Chapter 3, DCT does not rely on any specific properties of a particular DHT, such as the type of the hash function for example.

Each query q in the global query load L is assigned a probability p_q of being queried. We assume that the result set sizes of all queries from L are known: $|RS_q|,\ \forall q \in L$ denotes the number of documents in RS_q.

We denote the set of cached queries as $\Omega \subseteq L$. To store (to cache) all the result sets for all queries in Ω, the following global storage capacity is needed (measured in the number of documents): $S_\Omega = \sum_{\forall q_i \in \Omega} |RS_{q_i}|$.

Our goal is to utilize the available storage as efficiently as possible, which means to minimize the number of broadcasts or maximize the number of *cache-hits*. We denote a function $cachehit(q)$ as follows:

$$cachehit(q) = \left| \begin{array}{ll} 1, & \exists q' \in \Omega,\ s.t.\ q' \subseteq q; \\ 0, & otherwise. \end{array} \right.$$

Therefore, the cache optimization problem is to find the set Ω containing queries to be cached that maximizes the number of cache-hits:

$$\Omega = argmax \sum_{\forall q_i \in L} cachehit(q_i)\ p_{q_i},$$

having a storage constraint:

$$S_\Omega = \sum_{\forall q_i \in \Omega} |RS_{q_i}| \leq S_0.$$

The stated optimization problem is similar to the well-known 0/1 knapsack problem [Garey and Johnson 1979] (which is known to be NP-complete), applied to all queries from L. The increased complexity of the cache optimization problem compared to the knapsack problem is caused by the fact that we cannot assign constant profits to queries (items) due to the subsumption-related inter-dependencies between the queries. Furthermore, as the query load is dynamic, we are rather interested in a decentralized algorithm which leads to a quasi-optimal solution.

Indeed, each peer has to decide locally on a set of queries it caches to fill in its available storage. A peer pursues a greedy cache-selection strategy by deciding to cache queries such that their estimated *profits* are high. We define a max profit of the query q as: $profit_{max}(q) = \frac{g_q}{|RS_q|}$, where $g_q = \sum_{\forall q_i \subseteq q} p_{q_i}$ refers to the probability of the query q being utilized to answer any query from L if no other caches are available[2]. However, since there could be more then one cache capable of answering a given query due to the subsumption, the actual profit is lower and depends on the existing caches in the network.

An estimate of the query profit can be obtained from the statistics as $profit(q) = \sum_{\forall q_i \subseteq q} \frac{bfreq(q_i)}{|RS_q|}$, where $bfreq(q_i)$ is the number of broadcasts of q_i (because no caches subsuming q_i were found) observed *recently*. In this formula we can distinguish an absolute frequency $af_q = bfreq(q)$ of the query q being queried and a subsumption frequency

[2] A similar heuristic is used by [Baeza-Yates *et al.* 2007b]. In Chapter 6 we re-use it in the context of index pruning for Web search engines.

$sf_q = \sum_{\forall q_i \subseteq q} bfreq(q_i)$. The latter one counts all queries subsumed by q including q itself for the current state of the network. Obviously, $af_q \leq sf_q$. After we defined the subsumption frequency, the query profit can be finally expressed as:

$$profit(q) = \frac{sf_q}{|RS_q|}.$$

DCT peers perform local and isolated maintenance of the global query statistics: each peer has a global view (by listening to broadcasts) on the locally selected subset of queries it monitors. We restrict this monitored subset to the set of popular queries this peer used to submit in the past. The advantage of this mechanism is that approximate result set sizes are already known from the history. The statistics module counts recent absolute and subsumption frequencies for each query. When a peer caches a new query, it advertises the new cache in the meta-index as described above.

For every existing cache, similar statistics are maintained measuring its absolute and subsumption cache-hit values in order to evict it if more profitable caches were found.

Following the greedy strategy a peer can create a new cache if there is enough space available or the required amount of space can be released by evicting caches with lower profits. Hence, each peer locally selects the most profitable caches for the available local capacity. Therefore, if the query load is static, the greedy strategy ensures that the resulting cache-hit can only increase or remain the same.

Due to the multiple subsumption dependencies between queries, caching or evicting a cache might in some cases substantially influence statistics maintained for other related queries. We argue, that the presented strategy, though simple, gracefully adapts to the P2P network instability and changes in the query load. Indeed, the cost of adding a cache is only $O(\log N)$ overlay hops (needed to modify the meta-index), while evicting a cache causes only one extra message to be sent. In case of a peer failure all caches it stored or indexed become unavailable, causing broadcasting the associated queries. Thus, other peers will probably cache them if the profits are high enough.

Notice that a popular query q_0 might *not* be cached if it is associated with a large result set because its profit could be relatively low. In this case, DCT will react by caching popular derivatives of q_0 (queries subsumed by q_0) if needed. However, the meta-index would report a cache-miss for q_0 itself, and it would have to be broadcast every time. To solve this problem we suggest caching only the top-k results of the proper query q_0. Obviously, this "top-k" cache can not be utilized to answer any other query except for q_0. The profit of such a top-k cache can be estimated as:

$$profit_{topK}(q_0) = \frac{af_{q_0}}{k}.$$

Recall, that af_{q_0} denotes the absolute query frequency of q_0 being queried. The constant k reflects the maximum number of records the majority of the users would browse.

We choose the best type of cache for each query by comparing the estimations of profits calculated for the top-k cache and the full cache. If adding a new full cache fails, the second

attempt is made as top-k. A top-k cache can be switched back to a full cache by issuing a broadcast if the profit of the full cache is higher. Alternatively, when a full cache is about to be deleted it can be switched to the top-k cache instead.

The presented strategy facilitates the distributed selection of caches being constrained with the available storage capacity in the network and leads to a quasi-optimal solution with respect to minimization of the traffic consumption. Furthermore, utilizing top-k caches significantly reduces the number of cache-misses for the queries associated with large result sets and further decreases the traffic consumption.

4.2.3. Example

Let us illustrate the approach by an example. Initially, all queries are broadcast and each peer has to evaluate each query over its local document collection. A peer π joins the network and starts processing broadcasts and issuing its own queries. The peer π maintains statistics about the queries from its local query history, for example as shown in Table 4.2:

Multi-term query	$\|RS_q\|$	af_q	sf_q
cd	500	5	50
a	5000	98	100
ab	2000	21	23

Table 4.2. Example of query statistics maintained by a peer in the DCT approach.

The statistics table stores information about the most frequent queries from π's local query history. Recall that af_q and sf_q denote the absolute and subsumption frequencies respectively. $|RS_q|$ is estimated locally, since the result set for q was obtained by π before.

Assume the query *cd* is issued at π. The peer computes the full and the top-k profit estimates as $profit(\text{“}cd\text{”}) = \frac{sf_{cd}}{|RS_{cd}|} = 50/500 = 0.1$ and $profit_{topK}(\text{“}cd\text{”}) = \frac{af_{cd}}{k} = 5/250 = 0.02$, if $k = 250$ is chosen. Then it checks if there is enough storage space to cache *cd* as a full cache. If not, π compares $profit(\text{“}cd\text{”})$ with profits of already existing caches and evicts some of them if needed to store the result set of *cd*. Alternatively, it can consider caching the query as top-k, which would be the case for the query *a* for example.

Assume π is caching *cd*. It issues a broadcast and stores the obtained result set. To make the cache available for other peers in the network, π generates a key: $key = h(\text{“}c\text{”})$ (or, alternatively, $key = h(\text{“}d\text{”})$) and inserts the tuple $\{\text{“}cd\text{”}, address_\pi\}$ into the meta-index using the key key. The tuple is routed to the peer π_{key} responsible for the key and π_{key} stores the tuple. Assume also, that another peer caches a (top-k) result set for the query “*a*”.

Figure 4.2 shows how the 2-step query processing is performed. A peer π_{orig} submits the query $q =$ “*acd*”. First, the meta-index is searched for available caches. To do so, 3 messages containing q are routed to the peers responsible for $h(\text{“}a\text{”})$, $h(\text{“}c\text{”})$ and $h(\text{“}d\text{”})$ respectively (step 1). The peers π_a, π_c and π_d browse their meta-index tables and send back the lists of relevant

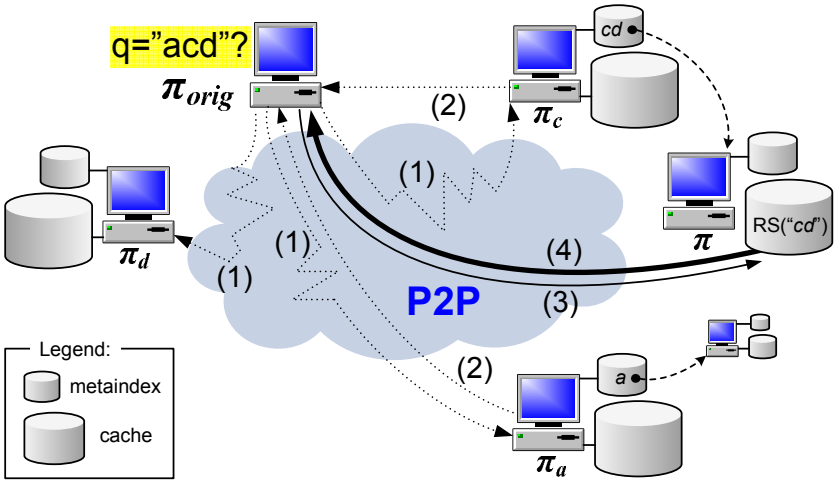

Figure 4.2. Query processing example.

caches (step 2). Note that since the query a is cached as top-k, it cannot be used to answer *acd*. Hence, the information that the query *cd* is cached at $address_\pi$ received from the peer π_c will be used. The originating peer requests π to answer the original query (step 3). The peer π responds back with the answer (step 4). In case no caches were found in the meta-index, a broadcast would be used to answer the query.

4.3. Load Balancing

Since our approach is based on caching popular queries, peers can suffer from certain load imbalances due to both non-uniform meta-index lookup requests and uneven cache utilization. In this chapter we show that load imbalance caused by these factors can be efficiently tackled without substantial performance degradation. In the following we discuss load balancing issues for both cases in detail.

4.3.1. Meta-Index Load Balancing

We show that due to the small size of the data stored in the meta-index and a certain randomization in the advertise operation, almost no explicit load balancing of the meta-index is necessary. However, peers that are responsible for the most popular terms can receive a large number of incoming requests, which can be avoided by handling such terms in a special way, *e.g.*, as discussed by [Cudré-Mauroux and Aberer 2002; Datta *et al.* 2007]. First, these terms are marked as popular and the index information involving them is moved to alternative locations if possible, since a query $q = t_1..t_n$ can be indexed on any of the n peers. Then, the

terms are advertised to the forwarding peers[3] as popular so these peers can take part of the load caused by the popular term. Thus, subsequent requests will not reach the original peer, but will be pruned on the way, leading to a better load distribution.

Indeed, our evaluations show, that only several top popular terms cause very high meta-index lookup load. Hence, the solution proposed above would split the load among neighboring peers, avoiding meta-index lookup hot-spots.

4.3.2. Cache Access Load Balancing

Balancing of the load caused by resolving queries from caches is more crucial due to the high traffic it creates to supply query results compared to the meta-index lookup. However, our evaluations show that only several top popular caches are accessed very often and cause serious load imbalance. Thus, standard replication mechanisms can be employed to relieve overloaded peers.

4.4. Experimental Results

In this section we report experimental results obtained by using our DCT simulator implemented in Java. The document collection used in the experiments is the collection of Wikipedia articles available at `http://www.wikipedia.org`. We used the 6GB XML dump of the core English Wikipedia from May 2006 that contains 3M pages. The dump is available at `http://download.wikimedia.org/enwiki/20060518`.

We used two real Wikipedia query logs from August and September 2004. Both of them have very similar properties, hence we summarize only those of the August trace. From the total of 4.6M queries, there are 1.3M unique queries. There are 0.5M queries occurring at least twice and 250K at least three times in the query log. The queries contain 160K unique terms while the average number of terms in a query is 2.6.

Both the Wikipedia document collection and the query logs were preprocessed by applying the Porter stemmer [Porter 1980]. Before performing the experiments we obtained the result set sizes for all the queries: first we built an in-memory single-term index for all terms appearing in the query logs and then we computed cache sizes for each query by intersecting posting lists for its terms.

4.4.1. Simulation Setup

The DCT simulator creates a number of peers with a predefined available cache capacity. It iteratively chooses random peers to generate queries and simulates the distributed query

[3] Forwarding peers are peers that have π_p in their routing tables, where π_p is the peer responsible for a popular term.

processing using the algorithms defined in Section 4.2. A query generator selects a real query from one of the query logs following the real query popularity distribution.

In most of the experiments we limit the available storage capacity per peer to 200K records and fix the top-k cache size to 250 records. As mentioned in Section 4.2.2, we monitor only recent query statistics, hence we selected a reasonable size of $200K$ broadcasts for the history window. In other words, the query statistics are maintained for the period which covers the last $200K$ broadcasts. Finally, a query can be cached only if it was already answered before, since its result set size has to be known.

In our experiments we measure three values: *CacheHit*, *SubsumHit* and *TopKHit*. *CacheHit* is the main measure that reflects the fraction of queries that were answered from caches, therefore the remaining $(1 - CacheHit)$ fraction of queries were answered using broadcasts. $CacheHit$ aggregates three different situations:

- A query q is answered from a top-k cache for q – we register this particular case as *TopKHit*.
- Alternatively, a query q produces a *SubsumHit* if it was answered using a full cache $RS_{q'}$ and the issued query q is different from the cached query q' (formally, $q' \subset q$).
- Finally, a query q can be answered from the full cache $RS_{q'}$ producing a *NormalHit*.

Obviously, $CacheHit = TopKHit + SubsumHit + NormalHit$.

4.4.2. Storage Capacity (Records)

In this experiment we explore how much capacity measured in *records* is needed to ensure a reasonable cache-hit rate. A record in this case refers to one entry in a posting list, *e.g.*, a posing list with x elements would consume x records. We vary the number of peers N, thus changing the overall network capacity as $N \times 200K$ records. Figure 4.3 plots the maximum *CacheHit*, *SubsumHit* and *TopKHit* achieved after the network converges to a stable state by processing 4.6M queries.

Figure 4.3 shows that high cache-hit values can be achieved with relatively low storage capacity, *e.g.*, DCT with 100 peers storing 200K records each, converges to 85% cache-hit. Another observation which we make from this plot is that top-k caches are extensively used when available storage space is limited (when the number of peers is below 20), whereas more and more queries are answered via subsumption as the capacity grows. Having high subsumption rate leads to a more robust network as we will see in the stress test experiment in Section 4.4.5.

When the DCT network contains 1,000 peers (marked with the asterisk) only 71% of the available capacity is utilized, thus the achieved cache-hit of 98% is the maximum for our experimental setting. The remaining 2% are singleton queries that are not subsumed by any cached query, so caching had no impact on them.

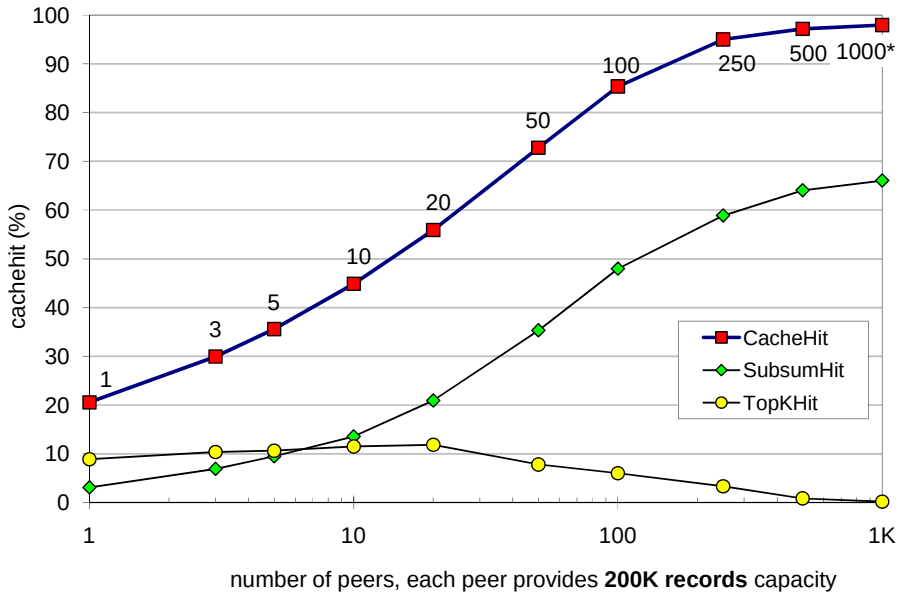

Figure 4.3. Max achieved *CacheHit*, *SubsumHit* and *TopKHit* for the different number of peers with **200K records** capacity each.

We achieved such a low cache-miss rate due to the high fraction of subsumption hits. Indeed, our simulations show that if peers have infinite capacity, but store only top-k caches, the maximum cache-hit that can be achieved is only 82%. This number can easily be obtained from the query load statistics. Recall, that out of 1.3M unique queries only 500K were repeated at least twice. Hence, caching has no impact on 800K singleton queries. Having 4.6M queries in total, these 800K queries are exactly the remaining 18%.

4.4.3. Storage Capacity (Bytes)

We repeat the experiment from Section 4.4.2 with one important difference: instead of using a notion of record we measure precisely how much space in bytes each cache would require. Thus, we can compute the real capacity of the network needed to ensure a reasonable cache-hit rate.

As in the previous experiment, we vary the number of peers N and fix the peer capacity to 20MB. The overall network capacity is therefore $N \times 20$MB. Figure 4.4 plots the maximum *CacheHit*, *SubsumHit* and *TopKHit* achieved after the network converges to a stable state by processing 4.6M queries.

Recall that each cache for a query q stores the result set $|RS_q|$ that contains document digests for (all or only top-k) documents matching q. We measured the size of the document digest $digest_d$ for each document d in the Wikipedia document collection. Thus, for a given cache we can approximate its size as: $|cache_q| = |h_q| + \sum_{\forall d \in RS_q}(|h_d| + |digest_d|)$, where $|h_q|$ and $|h_d|$ are the sizes of the cache header (64 *bytes*) and document header (64 *bytes*) respectively.

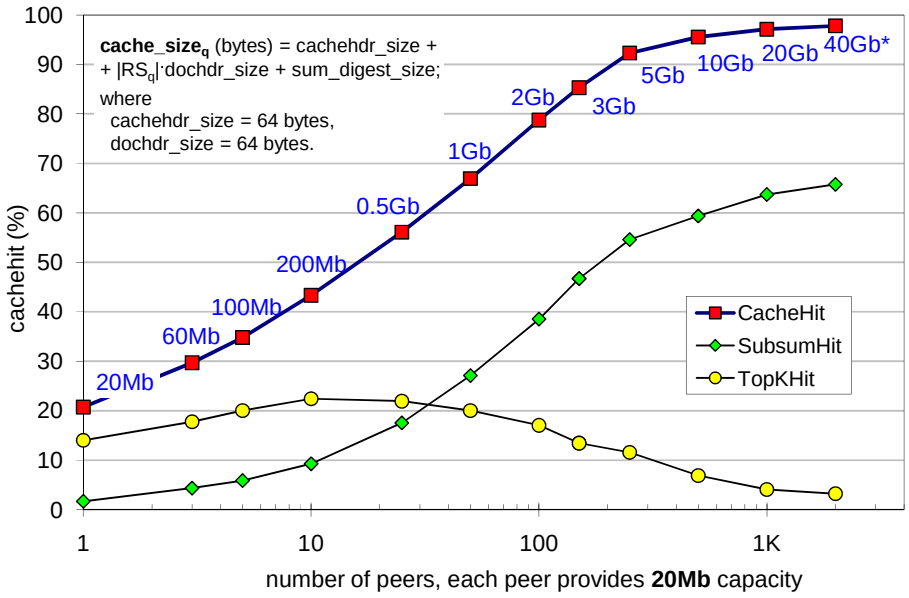

Figure 4.4. Max achieved *CacheHit*, *SubsumHit* and *TopKHit* for the different number of peers with **20 megabytes** capacity each.

Figure 4.4 shows that high cache-hit values can be achieved with relatively low storage capacity, *e.g.*, DCT with 100 peers providing 20MB of storage space each (2GB in total), converges to almost 80% cache-hit. Recall that the collection size is 6GB.

Another interesting observation can be made by comparing Figures 4.3 and 4.4: 20MB of storage could fit less than 200K records, thus in Figure 4.4 more top-k hits are registered when the number of peers is small. This happens, because the storage capacity is insufficient to fit some full caches. On the other hand, there are more subsumption hits with the same number of peers in Figure 4.3 because more storage is available. This example illustrates how DCT adapts to the current conditions of the network.

4.4.4. Traffic Consumption

The main goal of our simulation is to show that the proposed query-driven approach is a promising alternative to the standard single-term indexing techniques for P2P information retrieval. We will show that DCT reduces traffic consumption by two orders of magnitude when compared to the naïve approach, which distributes the standard global inverted index in the P2P network using term partitioning.

We implemented the naïve approach as follows. For each query we first eliminate all stop words contained in it (we used a list of 260 common English words). Then, we locate peers responsible for the remaining terms in the same way it is done by DCT. The query is answered by conveying a posting list between the peers responsible for the query terms. Posting lists are practically intersected along the way until reaching the final peer that produces the answer to

the query and sends only the top-10 records to the query originator (more records can be send on demand).

We implemented two variations of the naïve approach: *naïve-random* and *naïve-sort*. The first one chooses the terms and the responsible peers in a random order, whereas the second one sorts the terms according to the sizes of their posting lists in the ascending order and contacts the peers accordingly. We measure the traffic required to process a query as the number of records transmitted in the network. Note that assuming the term index is already available because it was produced beforehand during the indexing phase, the traffic required to process a query depends only on the posting list sizes of its terms. We computed the average traffic for the Wikipedia query logs and the data dump. Table 4.3 shows the obtained values after processing 4.6M queries.

Approach	August 2004	September 2004
naïve-random	37 370 rec./query	38 919 rec./query
naïve-sort	8 232 rec./query	8 903 rec./query
broadcast	7 920 rec./query	7 197 rec./query

Table 4.3. Average query traffic obtained with naïve approaches for the Wikipedia query load.

The size of the generated single-term index is 240M records. However, the index was build only for the terms extracted from the query load, whereas in practice this information is not available in advance.

The naïve approach performs poorly in terms of traffic consumption due to large posting list sizes which have to be transmitted. Surprisingly, Table 4.3 shows that the traffic consumption of the simple broadcast is even slightly better than the naïve-sort approach[4], although it requires propagating each query to all peers in the network.

Let us investigate the DCT performance in the dynamic setting. Figure 4.5.a shows how the *CacheHit*, *SubsumHit* and *TopKHit* values increase with the number of processed queries. The figure also plots the storage space utilization curve, which shows that all available space is almost fully utilized after processing 30K queries. Hence, the following cache-hit increase is achieved by proper selection of queries to cache. Finally, after enough statistics are gathered (after 200K broadcasts) the *TopKHit* starts decreasing while *SubsumHit* increases. It happens because some caches switch from top-k to full cache, based on the profit comparison.

Figure 4.5.b plots the average traffic per query generated by our approach (*DCT-All* curve). The *DCT-Indexing* curve shows the fraction of the *DCT-All* traffic, which was spent to create new caches. Traffic consumption is reducing rapidly as DCT converges. We also output the naïve and the broadcast traffic consumption for comparison. *DCT-All* curve is always below naïve and broadcast, moreover, the traffic consumed by our approach after DCT converged

[4] This happens mostly due to a relatively small number of peers used in the experiments. Also notice that our traffic computation ignores the control and routing messages sent between peers.

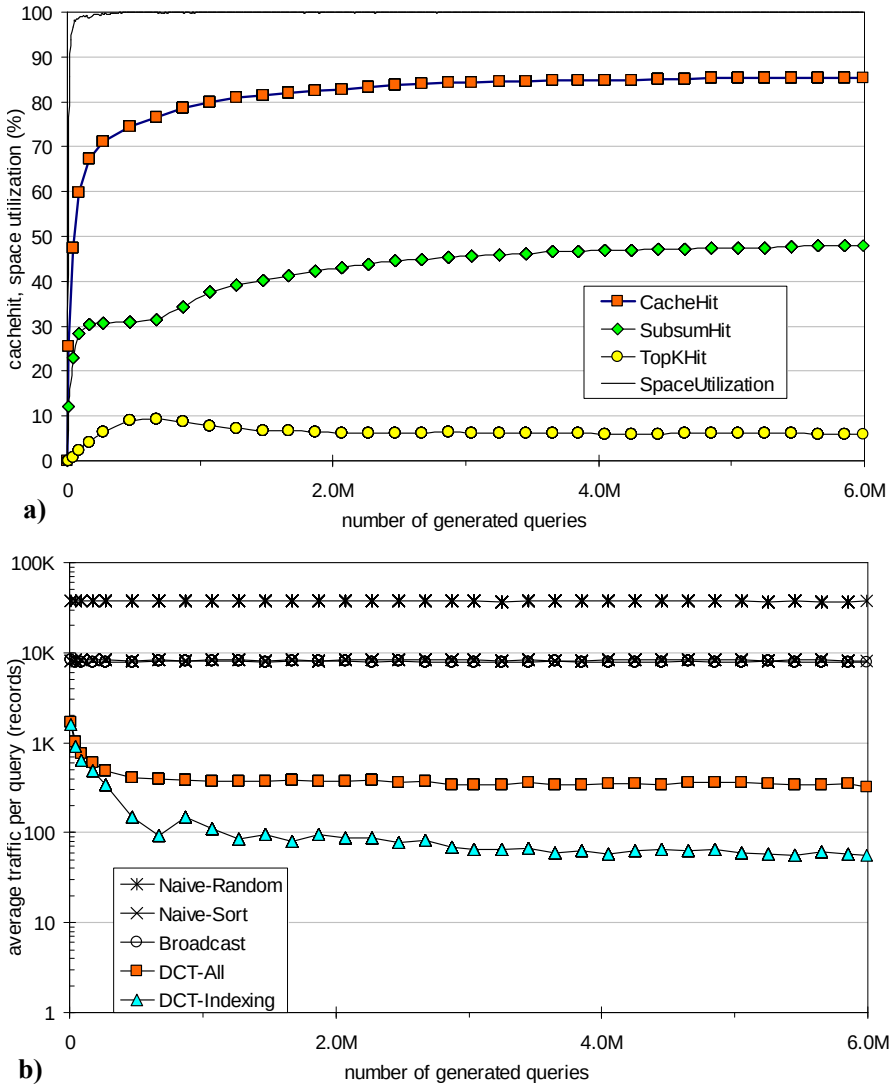

Figure 4.5. Cache-hit and traffic consumption for the network with 100 peers, 200K records per peer. a) Cache-hit; b) Traffic consumption.

(approx. 140 records/query) is two orders of magnitude lower than naïve-sort or broadcast. As we will see in Section 4.4.5 the DCT traffic consumption decreases even further when the cache-hit increases. The ideal value of 10 records/query would be achieved if all queries are answered from the cache.

Despite of low traffic requirements, DCT still has to broadcast the remaining $(1-CacheHit)$ fraction of queries that cannot be answered from the cache. This price is paid however, for a much smaller index size due to its adaptivity to the query load. In terms of latency DCT requires $O(\log N)$ time to answer a query from cache plus additional $O(\log N)$ in case the broadcast is required. The naïve approach requires additional time to transmit (possibly) large posting lists, which substantially increases the latency.

4.4.5. Stress Test

We performed a stress-test: in the first part of the test we generate queries from the August query trace and after 4.5M queries we switch to the September trace. Figure 4.6.a shows that DCT converges to the high cache hit rate of approx. 98%, slightly drops when the query load changes and converges again. The change is quite smooth because of the high subsumption utilization. Figure 4.6.b shows the traffic consumption during the stress test. It can be observed that the traffic consumption drops to relatively low values and slightly increases when the query load changes. Finally, with 98% cache-hit rate the traffic consumption reduces to approx. 75 records/query.

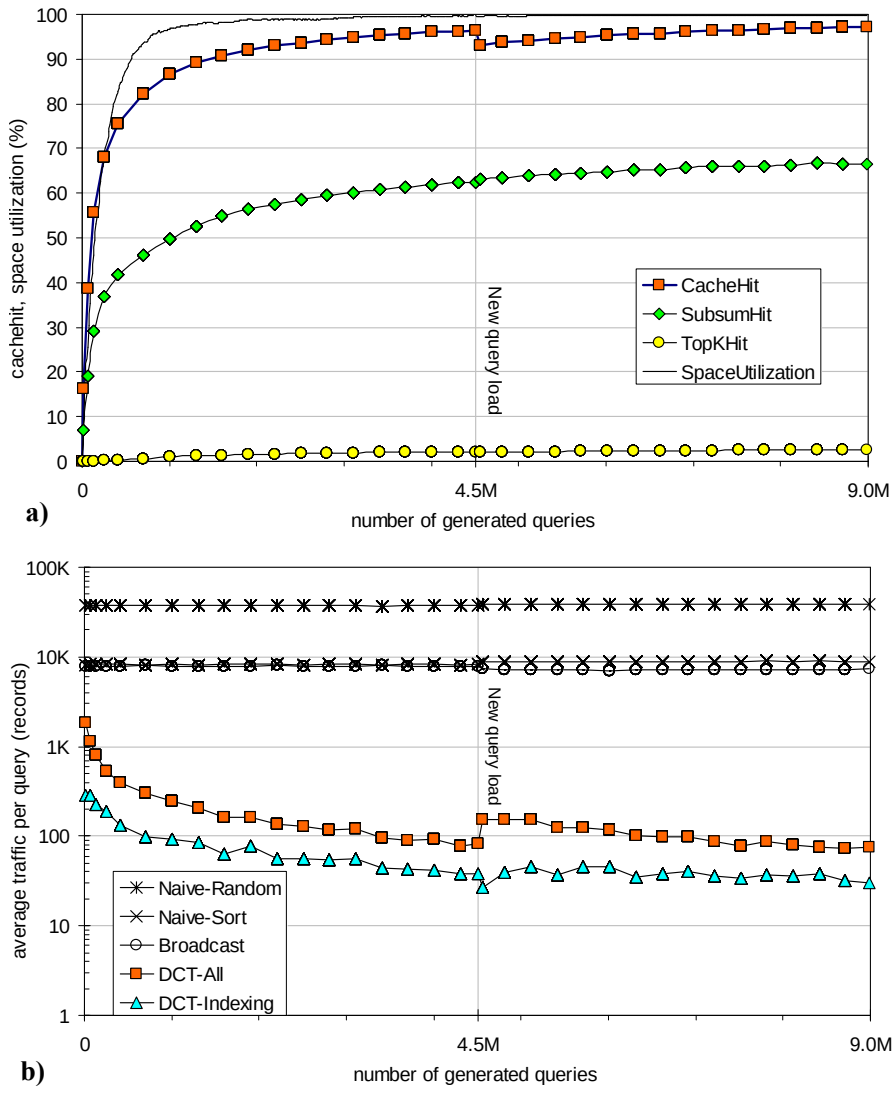

Figure 4.6. Cache-hit and traffic consumption while performing a stress test with 500 peers, 200K records per peer. a) Cache-hit; b) Traffic consumption.

4.4.6. Load Balancing

Figure 4.7 shows the peers' load obtained for the network consisting of 100 nodes and caused by answering queries from caches (Figure 4.7.a) and by executing meta-index lookups (Figure 4.7.b).

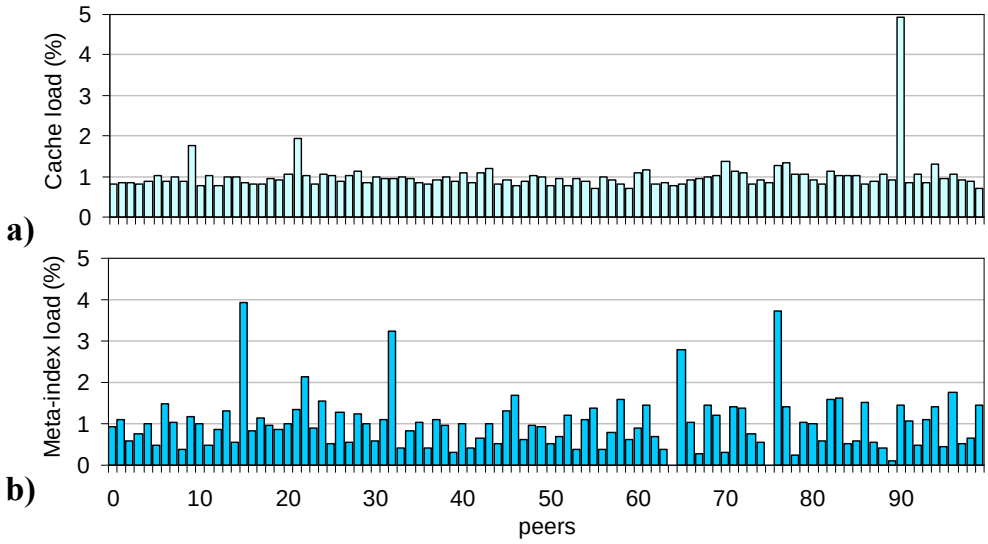

Figure 4.7. Load caused by a) cache accesses and b) meta-index lookups in the network of 100 peers.

It might seem that popular caches cause huge load imbalance. However, if a user requests only top-10 items and taking into account unrestricted cache placement, which depends only on the peers' querying activities, the load is distributed almost evenly as it can be observed in Figure 4.7.a. The only problem is caused by several top-popular queries that create very heavy load. A native DHT replication mechanism can be used to solve this imbalance. Also notice that in our experimental setting the Wikipedia query log could be the culprit: the most popular query "wikipedia" occurs very often (in 2.5% of the cases) and explains the spike on Figure 4.7.a.

The meta-index service exhibits a certain load imbalance as shown in Figure 4.7.b, however it serves only index lookups that do not require large traffic transfers. Moreover, TCP/IP connections to the neighbors are maintained alive by the DHT, hence, the imbalance caused by the meta-index has low impact compared to the cache imbalance. Additionally, the mechanisms we proposed in Section 4.3.1 would help avoiding hot spots with low overhead.

4.4.7. Term Combinations vs. Queries

The last experiment we performed suggests that the cache-hit values reported before can be further improved by considering all possible term combinations as candidates for caching instead of using only real queries. We measure the *CacheHit* values for 100 peers with 20MB

capacity each as we did before with one difference: each peer can cache not only full queries but also any combinations of terms contained within past queries. The benefit of such a modification is clear – the number of all possible term combinations that can be cached increases significantly enabling a better selection of the most profitable caches.

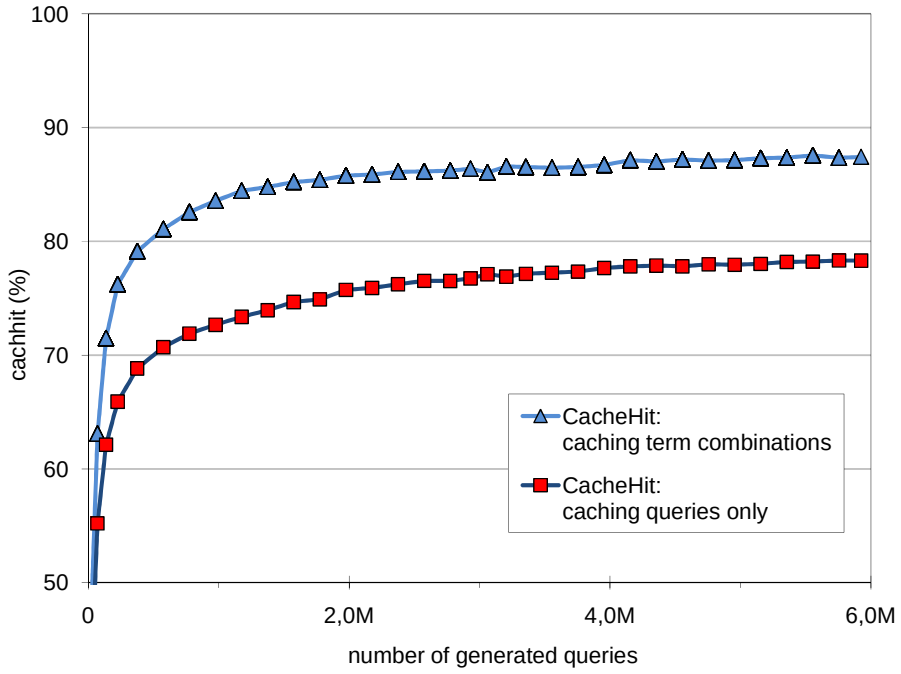

Figure 4.8. *CacheHit* for 100 peers with **20 megabytes** capacity each: caching arbitrary term combinations vs. caching queries only.

Figure 4.8 shows the *CacheHit* increase of nearly 10% in our setting with 100 peers with 20MB capacity each. This indicates that very profitable combinations might not be queried alone, but often included in queries along with other terms. We explore the option of term combinations caching in more details in Chapter 5. Similar results for XPath query caching were recently reported by [Lillis and Pitoura 2008].

4.5. Conclusions

In this chapter we presented a novel query-driven indexing strategy for multi-term query processing with structured P2P networks. In our approach, called Distributed Cache Table (DCT), each peer runs a greedy algorithm leading to a quasi-optimal network-wide cache selection that maximizes the global cache-hit rate. DCT relies on the subsumption relation between queries while selecting a cached result set to resolve a query.

We performed an extensive experimental evaluation on real data and query traces that confirms the feasibility of our approach. The results showed two orders of magnitude reduction

in traffic consumption compared to the naïve single-term indexing approach.

We believe that DCT can be applied to the broader class of *conjunctive queries*. Such a query is expressed by a conjunction of atomic predicates: $q = a_1 \& a_2 \& .. \& a_n$, where the atomic predicate structure is application-specific, *e.g.*, for the studied class of multi-term queries each atomic predicate is a natural language term.

We envision DCT to be a perfect solution for distributed digital libraries, where DCT offers an easy to use indexing service without any central coordination point in a relatively stable and reasonably small network.

However, it is also clear that caching full posting lists even for combinations of terms (queries) becomes very expensive when the document collection size reaches Web scale. Another issue of the DCT approach is that it has to maintain document digests to enable local post-processing and make use of query subsumption. Clearly, document digests consume significant amount of storage space and traffic. Chapter 5 presents a different Query-Driven Indexing approach (simply called the *QDI approach*) that addresses these problems by using truncated versions of posing lists and keeping only document *id*s in the posting lists.

Chapter 5

Scalable Web Text Retrieval with a Query-Driven Index in a Peer-to-Peer Network*

5.1. Introduction

In this chapter we continue to explore query-driven indexing methods for P2P text retrieval. The Distributed Cache Table (DCT) approach presented in Chapter 4 provides an elegant distributed caching solution, but requires indexing and maintenance of full posting lists for popular combinations containing relatively big document digests. Thus, with a Web-scale document collection DCT would suffer from substantially reduced cache-hit rates because fewer popular (and possibly associated with very large caches) combinations will be cached.

In this chapter we present an alternative query-driven indexing/retrieval strategy for efficient full text retrieval from a *Web-scale* document collection distributed within a structured P2P network. Our novel indexing strategy is based on two important properties:

1. the generated distributed index stores posting lists for *carefully chosen indexing term combinations* that are frequently present in user queries, and
2. the posting lists containing too many document references are truncated to a *bounded number of their top-ranked elements*.

These two properties guarantee acceptable latency and bandwidth requirements, essentially because the number of indexing term combinations remains scalable and the posting lists

* The material presented of this chapter aggregates several research papers published in the proceedings of:
- the 16th International World Wide Web conference (WWW'07, poster track) [Skobeltsyn *et al.* 2007a],
- the 2nd International Conference on Scalable Information Systems (Infoscale'07) [Skobeltsyn *et al.* 2007b],
- the 30th International ACM SIGIR Conference (SIGIR'07) [Skobeltsyn *et al.* 2007c].

The journal version is available in [Skobeltsyn *et al.* 2009]. The approach is implemented in the AlvisP2P prototype [Luu *et al.* 2008], Website – `http://globalcomputing.epfl.ch/alvis`.

transmitted during retrieval never exceed a constant size. An index update mechanism efficiently handles adding of new documents to the document collection. Thus, the generated distributed index corresponds to a constantly evolving *query-driven indexing structure* that efficiently follows current information needs of the users and changes in the document collection. We will show that the size of the index and the generated indexing/retrieval traffic remains manageable even for Web-size document collections at the price of a marginal loss in precision for rare queries. Our theoretical analysis and experimental results provide convincing evidence about the feasibility of the query-driven indexing strategy for large-scale P2P text retrieval.

As we already discussed in Chapter 2, extensive bandwidth consumption has been identified as one of the major obstacles for the adoption of peer-to-peer (P2P) technology in the field of information retrieval (IR). Studies such as [Li *et al.* 2003; Zhang and Suel 2005] have shown unscalable traffic requirements for Web-size document collections, even when sophisticated protocols are used to reduce retrieval costs. Recall that query processing with a term-partitioned distributed index requires intersection of posting lists that can physically reside on different nodes of the network, causing substantial latencies and traffic consumption. Instead of indexing single terms found in the document collection, which might lead to potentially very large posting lists and thus unscalable bandwidth consumption, we focus our research efforts on building distributed indexing structures based on term combinations.

The rationale behind this idea is that due to a high degree of distribution in a P2P network it is more efficient to store the index in a large number of small blocks instead of a small number of large blocks. For the P2P-IR scenario we reformulate the last sentence in a more concrete way: it is easier to index a large number of term combinations associated with small posting lists than a small number of terms with large posting lists. Notice that we implicitly exploited this idea already in Chapter 4 while designing the Distributed Cache Table approach.

Alternatively, the approach based on indexing with Highly Discriminative Keys (HDK) [Podnar *et al.* 2007] follows the idea of using term combinations as well. In Section 2.2.3 we already described the basic properties of the HDK approach. Recall, that it relies on indexing with terms and term combinations (called *keys*) that occur in at most DF_{max} documents, where maximal document frequency DF_{max} is a parameter of the model. In the HDK approach keys with a document frequency exceeding the predefined DF_{max} threshold are associated with truncated posting lists, only storing the top-DF_{max} ranked documents and are possibly expanded into several larger keys (*i.e.*, keys consisting of more indexing terms) with smaller document frequencies. The scalability analysis of the HDK approach [Podnar *et al.* 2007] has shown that the number of generated keys grows linearly with the number of documents, which is acceptable under a reasonable assumption that the ratio between the total number of documents and the total number of peers in the network remains bounded.

However, we have observed that the HDK approach generates a large number of keys that are never or rarely used in queries. Indeed, as the keys are generated only on the basis of their document frequencies, their popularity (and thus practical usefulness) is not taken into account.

Obviously, the creation and maintenance of such superfluous keys causes substantial consumption of both bandwidth and storage, which represent valuable resources in large-scale networks.

In this perspective, we started exploring ways to eliminate superfluous keys from the index such that the retrieval quality remains acceptable. As a result, we designed the *Query-Driven Indexing* (*QDI*) strategy, which extends the HDK indexing mechanism, by taking into account the popularity of term combinations appearing in user queries. The QDI approach uses query statistics to filter out superfluous keys which results in a substantial reduction of the generated index size compared to the HDK approach. As a consequence, the quality of the answer obtained for a given query depends on the popularity of the term combinations it contains. We have observed that this leads to only a marginal loss in retrieval quality, as the indexing structure constantly evolves and reacts to changes in the query distribution.

In this chapter we present the detailed description of the QDI approach. The rest of the chapter is organized as follows. We explain the indexing/retrieval model in Section 5.2 followed by the algorithms overview in Section 5.3. We then present the scalability analysis in Section 5.4 and the experiments in Section 5.5. We describe the AlvisP2P prototype that implements QDI in section 5.6 and provide conclusions in Section 5.7.

5.2. Distributed Query-Driven Indexing/Retrieval

In this section we introduce our P2P-IR framework and outline indexing/retrieval routines. We then present an indexing on-demand mechanism that is used to activate new multi-term keys detected as popular and to generate the associated posting lists. Finally we discuss how to maintain the contents of the index up-to-date w.r.t changes in the document collection.

5.2.1. P2P Global Index

Let us consider a structured P2P network with N peers $P_i,\ 1 \leq i \leq N$, and a possibly very large document collection $\mathcal{D}$, consisting of $|\mathcal{D}|$ documents $d_j,\ 1 \leq j \leq |\mathcal{D}|$. $T_\mathcal{D}$ is the set of all indexing single terms in $\mathcal{D}$ and $|T_\mathcal{D}|$ denotes the number of terms in $T_\mathcal{D}$.

In addition, we assume that a large query log L is available, where each query $q \in L$ is a set of terms. The set of all terms present in the query log is denoted by T_L, and the intersection $T_L \cap T_\mathcal{D}$ thus corresponds to the query terms producing non-empty results.

In the P2P network, each peer P_i stores a fraction of the global document collection $\mathcal{D}$, denoted by $\mathcal{D}_i$, and builds a local index for $\mathcal{D}_i$. At the same time, P_i contributes to store and maintain a fraction of the global distributed index $\mathcal{I}$ that associates keys with references to documents in $\mathcal{D}$.

Definition 5.1. A **key** k refers to an indexing term or a combination of indexing terms. The standard stop-word elimination and stemming procedures [Porter 1980] are applied when generating a key.

A posting list $\rho(k)$ associated with a key k is the list of references to documents that contain k: $\rho(k) = \{d_j \in \mathcal{D} \mid k \in d_j\}$, where $|\rho(k)|$ corresponds to the document frequency of k, denoted as $df(k)$. In addition, a set of statistical values such as term frequency, document size, *etc.*, is stored in the posting list for each pair (d_j, k), $d_j \in \rho(k)$. These values are used to compute the final relevance score of the document for any query containing k as well as the score $r(d_j, k)$ that determines the rank of the document in the posting list. Various models can be used to compute the relevance scores. Currently, we are using the top performing BM25 relevance computation scheme [Robertson *et al.* 1992]. Notice, however, that any other relevance computation scheme could be used instead, provided that the required global statistics for relevance computation are available in the P2P network.

Definition 5.2. A **truncated posting list** (*TPL*) $\tau(k)$ associated with a key k refers to the DF_{max} best-ranked document references in the posting list $\rho(k)$, where DF_{max} is a parameter of our model, corresponding to the maximal size of a TPL.

By construction, $|\tau(k)| \leq DF_{max}$, and, obviously, $\tau(k) = \rho(k)$ when $|\rho(k)| \leq DF_{max}$.

Definition 5.3. The **usage frequency** $qf(k)$ of a key k is a metric indicating the global popularity of the key k in the query log L. In the simplest case, $qf(k)$ can be the query frequency of k, which is incremented each time a new query containing k is processed.

N	Number of peers in the P2P network
$\mathcal{D}$	Document collection
$T_{\mathcal{D}}$	The set of all unique terms in $\mathcal{D}$ (the vocabulary)
L	Query log
$\mathcal{I}$	Global index
k	Single-term or multi-term key
$\rho(k)$	Full posting list associated with the key k
$r(d_j, k)$	Relevance score of a document d_j with respect to k
$\tau(k)$	Truncated posting list (TPL) for k
$df(k)$	Document frequency of the key k, which equals $\|\rho(k)\|$
$qf(k)$	Usage frequency of the key k
s_{max}	The maximum number of terms that any key can contain
DF_{max}	The maximal size of a TPL
QF_{min}	The minimal query frequency in order to activate a key

Table 5.1. Main notations of Chapter 5.

Currently, we use frequency counters that maintain usage frequencies within a predefined time interval and therefore enable a timely reaction to changes in the query popularity distribution.

In our framework we introduce three types of index items that are used to store the indexing information in the P2P network. Posting lists for all *single* terms found in the document collection $\mathcal{D}$ are stored in *basic index items*. Popularity statistics for selected multi-term combinations is maintained in *candidate index items* and, finally, truncated posting lists (TPLs) for the most popular combinations are stored in *active index items*. The formal definitions are given below:

Definition 5.4. A **basic index item** for a *single-term* key k, $|k| = 1$ is the pair $(k,\ \rho(k))$ associating k with its full posting list $\rho(k)$. The corresponding TPL $\tau(k)$ can always be locally obtained from the full posting list $\rho(k)$.

Definition 5.5. A **candidate index item** for a multi-term key k is the pair $(k,\ qf(k))$ associating k with its usage frequency $qf(k)$. The candidate index item is created for the key k and inserted into the global index $\mathcal{I}$ iff:

- k contains from 2 to s_{max} terms: $2 \leq |k| \leq s_{max}$, where s_{max} is a parameter of our model (*size filter*).
- the document frequency $df_{k'}$ of each sub-key k' of the key k of size $|k| - 1$ is above DF_{max} and the corresponding basic or active index item for k' is already stored in the global index. $\forall\ k' \subset k,\ |k'| = |k| - 1 : df_{k'} = |\rho(k')| > DF_{max}\ \&\ \tau(k') \in \mathcal{I}$ (*redundancy filter*).

Definition 5.6. An **active index item** for a multi-term key k is the triple $(k,\ qf(k),\ \tau(k))$ associating k with its usage frequency $qf(k)$ *and* the TPL $\tau(k)$. An existing candidate index item for a multi-term key k is *activated* (*i.e.*, its status is changed from candidate to active) iff:

- k is *popular*: $qf(k) \geq QF_{min}$, where QF_{min} is a parameter of our model (*popularity filter*).

Within such a setup, the global distributed index $\mathcal{I}$ maintains:

- a set of basic index items for all single-term keys found in the document collection $\mathcal{D}$,
- a set of candidate and active index items generated for the multi-term keys extracted from the query statistics.

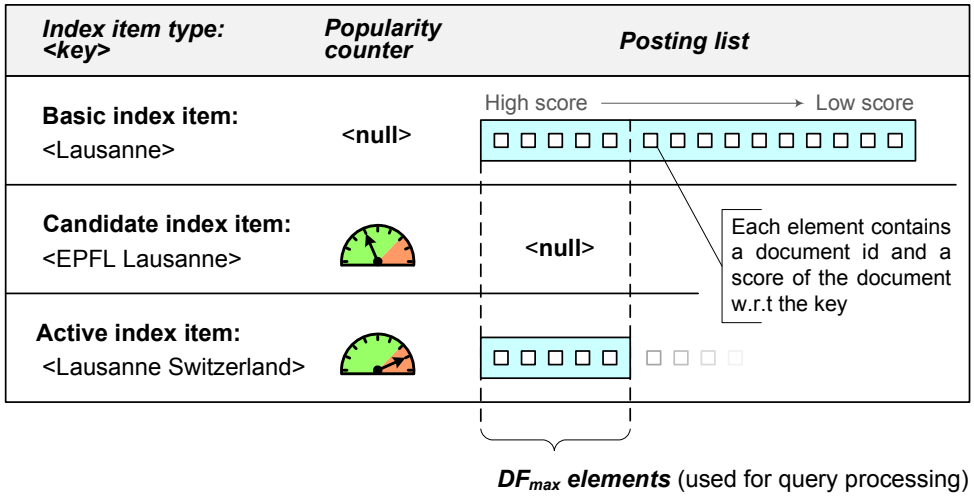

Figure 5.1. Example of index item types.

Definitions 5.4–5.6 are illustrated in Figure 5.1 that shows examples for all three types of index items. Notice that each single-term key found in $\mathcal{D}$ is always associated with a basic index item (Figure 5.1-a). A multi-term key k can be either 1) not present in the index, or 2) associated with a candidate index item (Figure 5.1-b), or 3) associated with an active index item (Figure 5.1-c).

We call a multi-term key k a candidate key, if the global index contains a *candidate* index item for k. Similarly, we say that a key k is indexed, if the TPL $\tau(k)$ can be obtained from the index, *i.e.*, an *active* or a *basic* index item can be found for k. Notice also that a multi-term key can be indexed only if it passed all three filters (size, redundancy and popularity) defined above (see Definitions 5.5, 5.6).

The global index is distributed over the peers such that the fraction of the index under the responsibility of a peer P_i is exactly the set of index items associated with the keys that are allocated to P_i by the Distributed Hash Table (DHT) built on top of the P2P network. In such a DHT, the peer responsible for a given key can be uniquely determined by applying a globally known hash function, thus, ensuring balanced placement of the indexing information. An efficient and self-organizing communication protocol enables any peer to route a message to the peer responsible for a given key in $O(\log N)$ overlay hops, where N is the total number of peers in the network. The details of such a protocol can be found in Chapter 2.

Locally each peer P_i is responsible for the following complementary tasks:

- P_i ensures that its local document collection $\mathcal{D}_i$ is properly represented in the global distributed index $\mathcal{I}$.
- P_i maintains its fraction of the global index $\mathcal{I}_i$[1]. In particular, it takes care that the

[1] Notice that the fraction of the global index maintained by P_i has no a priori reason to be related to the local document collection stored at P_i.

usage frequencies are updated during query processing. Based on this information, P_i can decide either to activate candidate index items or to deactivate active index items.

- While processing a query q, P_i interacts with the global P2P index in order to retrieve available relevant TPLs from the distributed index. If necessary, P_i can request the corresponding peers to create new candidate index items for the keys contained in q.

A more detailed description of the above-mentioned indexing and retrieval tasks is given below. The formal description of the corresponding algorithms is presented in Section 5.3.

5.2.2. Indexing/Retrieval Mechanisms

The goal of distributed indexing is to generate and maintain a suitable set of index items, associated with the corresponding TPLs, for any given global document collection $\mathcal{D}$ distributed over N peers and the current query popularity distribution. Since the indexing process is computationally intensive, peers share the indexing load, and collaboratively build the required distributed index.

First, the peers build the *basic single-term index* that contains basic index items ($\{t, \rho(t)\}$ tuples) for all single terms in $T_{\mathcal{D}}$. Each peer P_i performs indexing of its local document collection $\mathcal{D}_i$ and inserts all single-term keys associated with their local posting lists, into the P2P network. As a result, a *full posting list* is acquired for each single term $t \in T_{\mathcal{D}}$ maintained at the peer P_t responsible for t. Recall that $\tau(t) \subseteq \rho(t)$, *i.e.*, the TPL for a term t can be locally generated from its full posting list. To ensure bounded bandwidth consumption only (short) TPLs are used for retrieval. Thus, the basic index enables the processing of any query, possibly with a degraded retrieval performance due to the loss of information caused by the TPL truncation. Full posting lists are used for key activations and index updates as described in Sections 5.2.3 and 5.2.4 respectively.

After the basic index is built, the subsequent indexing process[2] is fully driven by the query statistics, and is performed in parallel with retrieval. More precisely, as soon as a peer P receives a new query q, it starts to explore the lattice of the query term subsets (hereafter called the *query lattice*), in decreasing subset size order starting with the query itself[3]. An example of a query lattice is given in Figure 5.2, which shows 3 potential scenarios for query processing. For each of the explored lattice nodes q' (hereafter called the *query keys*), the querying peer P requests from the peer P' responsible for q' the TPL associated with q'.

Each peer P' that receives such a request attempts to increment the usage frequency of q' in the corresponding index item. If no index item exists for q', no extra action is taken. Additionally, as soon as a candidate key q' becomes popular, P' initiates the key activation process by triggering the on-demand indexing mechanism for q' (see Section 5.2.3). As a result

[2] It can also be viewed as a form of distributed caching on top of the basic single-term index.

[3] Notice that if the query has more than s_{max} terms, the lattice exploration starts from the term subsets of size s_{max}.

of the on-demand key indexing, the candidate index item associated with q' will acquire a new global TPL and thus become an active index item that can be used for subsequent query processing. For example, in Figure 5.2-b, a candidate index item is stored for the key *bc*, and, as soon as its popularity reaches the QF_{min} threshold, it is activated as shown in Figure 5.2-c.

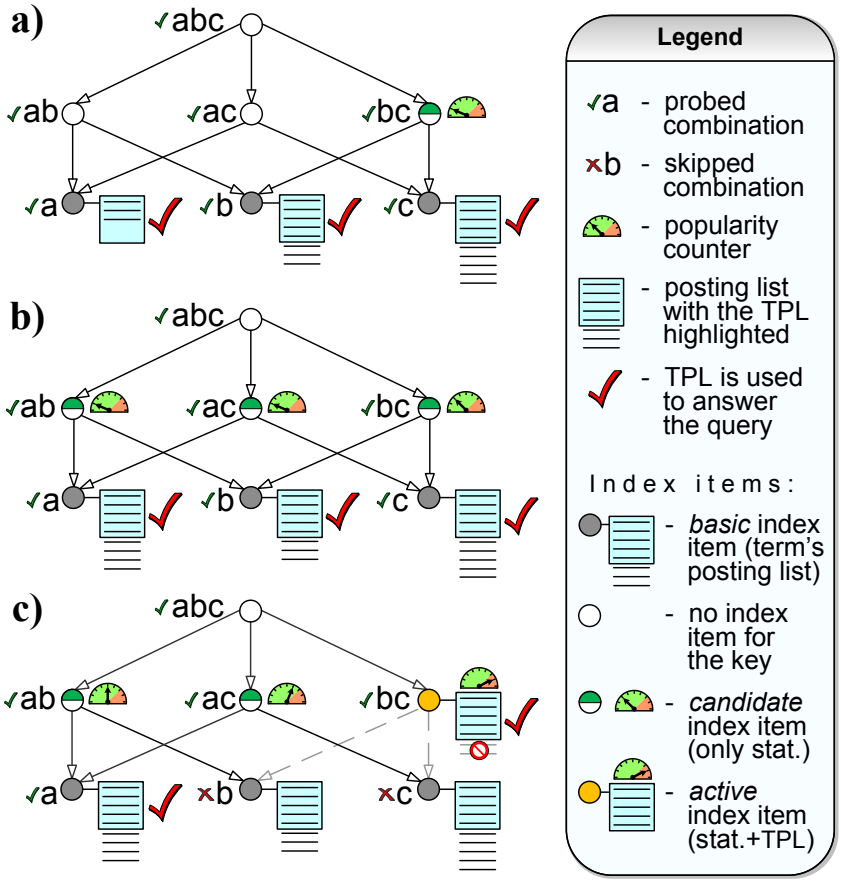

Figure 5.2. Possible scenarios when processing the query *abc* if: a) the posting list for a is truncated, while posting lists for b and c are not; b) the posting list for a is also truncated; c) additionally the key *bc* is indexed.

If q' is indexed, the peer P' sends back to the querying peer P the content of the TPL $\tau(q')$ associated with q'. When P receives the TPL, it stores it locally, and the part of the query lattice dominated by q', *i.e.*, all the query keys contained in q', are excluded from the subsequent lattice exploration. For example, for the query lattice shown in Figure 5.2-c, if a TPL associated with the key *bc* is retrieved from the P2P index, the query keys b and c are not further explored. Thus, the top-down query lattice exploration can lead to two mutually exclusive terminal situations:

- At least one query key associated with a *non-truncated* posting list is reached offering an exhaustive list of potential answers to the query. For example, the key a in Figure 5.2-a

is associated with an exhaustive posting list ($|\rho(a)| \leq DF_{max}$) that can be used to answer any query containing a, *e.g.*, *abc*.

- A *cut* of query keys (see Definition 5.7) associated with *truncated* posting lists is reached. If none of the multi-term query keys is popular enough to be associated with an active index item, the cut will consist of all the single-term keys contained in the query (see Figure 5.2-b). Otherwise, the cut can consist of keys of different sizes (for example, in Figure 5.2-c keys a and bc form the cut for the query *abc*, and the associated TPLs are used for query answering).

Definition 5.7. In a subset lattice a **cut** is a set of nodes N_i, s.t.: 1) the union of all the nodes dominated by N_i is equal to the top node of the lattice; and 2) none of the nodes in the cut dominates any other node from the cut. For example the set of nodes a and bc in Figure 5.2-c is a cut because: 1) $a \cup bc = abc$, and 2) $a \nsubseteq bc,\ bc \nsubseteq a$.

When either of the two terminal situations is reached, the query lattice exploration stops and all the retrieved TPLs are used by the querying peer for postprocessing. More precisely, the peer produces their union, re-ranks all the resulting documents d_j with respect to the original query q (*i.e.*, it computes the relevance scores $r(d_j, q)$ using the values stored in the postings), and presents the top-ranked document references to the user as the result for the submitted query q.

For example, Figure 5.2 shows the processing of the query *abc* with three different states for the global index. Notice that the tick sign highlights the TPLs that are collected by the querying peer and are used to produce the final result for *abc* in each case. The top-k results obtained for the query *abc* in the case 5.2-c can potentially be of better quality than the ones in 5.2-b due to the fact that an extra TPL for the key bc is available. In the case 5.2-a, the quality of the result is already maximal because the size of the posting list for a is below DF_{max} and hence it contains all possible document references relevant for a, and therefore for *abc*.

Finally, as a consequence of the query lattice exploration, the querying peer can discover new candidate multi-term keys that have to be created. According to Definition 5.5, these candidate keys belong to the set of *immediate ancestors* of all identified query keys that are: 1) indexed, and 2) associated with truncated TPLs[4]. The peer then simply requests the creation of all such candidate index items, provided that they are not already stored in the global index. Recall that the created candidate index items will only maintain their usage frequencies, and can be activated later in case they become popular.

To summarize, the processing of new queries leads to key activations and hence to the generation of new TPLs, which, in turn, increases the retrieval quality for subsequent queries.

[4] In other words these candidate keys belong to the set of immediate ancestors of all the nodes in the cut.

Obsolete active keys that become unpopular over time due to changes in the user query distribution can also be deactivated, thus constantly adapting the set of active index items stored in the global index to the current users information needs.

5.2.3. On-Demand Indexing Mechanism

When a multi-term key k is activated, the on-demand indexing mechanism is executed by the peer P responsible for k to generate the global TPL $\tau(k)$ containing top-DF_{max} document references found in the global document collection for a given k. Notice that after the TPL is generated, a query-driven index update mechanism (see Section 5.2.4) is used to maintain it up-to-date with respect to the document collection changes (*e.g.*, the addition of a new relevant document might require changing of existing TPLs).

However, in order to activate a new key, top-DF_{max} relevant documents (ranked w.r.t the key) currently present in the global document collection have to be identified. As all peers could potentially hold documents containing k, a naïve approach would be to broadcast an indexing request containing k to the whole network. P would then collect the answers and generate the global TPL. Such a naïve approach is obviously quite expensive in terms of bandwidth consumption and can also lead to load balancing problems. Nevertheless, our initial solution described in [Skobeltsyn *et al.* 2007b] was based on a carefully optimized version of broadcast, which uses a special P2P-level multicast and various piggybacking techniques to reduce bandwidth consumption. However, the broadcast-based solution might still not scale well with the network size.

A more bandwidth-efficient solution is to utilize the full posting lists stored in the basic index items for all single-term keys. Then the on-demand indexing can be carried out in a conventional way by intersecting the full posting lists of all single terms contained in the key that is being activated. Any distributed set intersection algorithm, *e.g.*, based on Fagin's Threshold Algorithm [Fagin *et al.* 2001], can be used.

The substantial difference compared to the standard single-term indexing approach is that the intersection operation is not performed on a per-query basis (*i.e.*, frequently), but is executed only once when a new key is activated. Moreover, as the indexing latency is less crucial than the retrieval latency, we can tolerate a certain delay of key activation.

5.2.4. Updates in the Query-Driven Index

In order to keep the contents of the index up-to-date, TPLs associated with active index items have to be constantly refreshed such that new documents can be found shortly after they have been added to the global collection.

Updating the query-driven index is a challenging task as the set of indexed keys relevant to a document is unknown at indexing time. A naïve approach would be to probe all possible keys extracted from the document and to update the corresponding TPLs. Obviously, even if

updating posting lists associated with single terms is possible, such a strategy fails miserably given the number of possible term combinations of size 2 and larger.

To explore the setting in more detail we performed the following experiment. We used a large corpus of 17M AOL queries (see Section 5.5 for more details) to build a query-driven index $\mathcal{I}$. We discovered that the set of keys $M(d)$ that match a document d ($M(d) \subseteq \mathcal{I} \mid \forall k \in M(d) : \ d \ni k$) is typically large, normally reaching the order of thousands of keys. However, for a large (Web-size) document collection, the set of keys $U(d)$ for a document d that should be updated ($U(d) \subseteq M(d) \mid \forall k \in U(d) : \ TPL(k) \ni d$) is typically very small. The reason for this is that there are usually only few keys that will update their TPLs (top-DF_{max} results) with the new document reference d, but there are many keys matching the document whose top-DF_{max} document references are ranked higher than d. Figure 5.3 illustrates this observation: all keys that match the document $M(d)$ do not have to be updated except for the small subset $U(d)$, $U(d) \subseteq M(d)$, where each key from $U(d)$ will include the new document in its top-DF_{max} results.

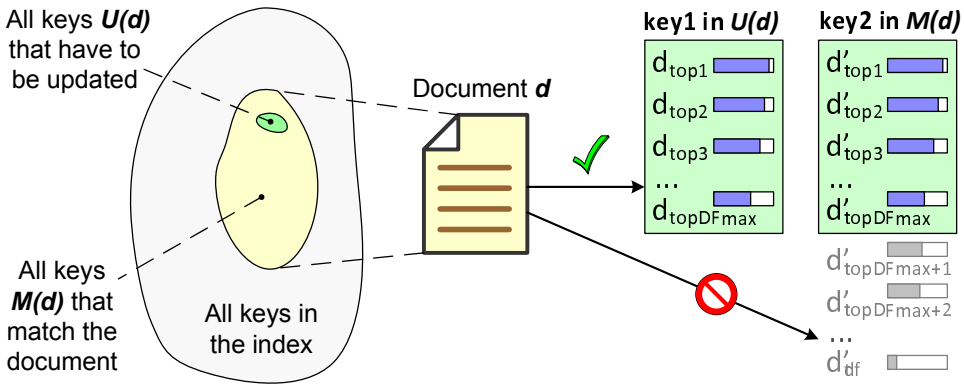

Figure 5.3. Updating the distributed index for a new document.

For example, we took `http://globalcomputing.epfl.ch/alvis` as a test document (the main page of our AlvisP2P prototype Web site) and used the AOL query log (see Section 5.5.1) to identify popular multi-term keys that are contained in the document (*i.e.*, the $M(d) \setminus T_d$ set). We crawled Google's top results for each key in $M(d)$ and checked whether the original document is contained within the top-200 results. We obtained Table 5.2 for the AlvisP2P Web page.

The digest size (the number of unique terms in d) $\lvert T_d \rvert$	96
The number of indexed keys matching the document $\lvert M(d) \rvert$	2482
The number of indexed keys that have to be updated $\lvert U(d) \rvert$	5

Table 5.2. Indexing statistics for the sample page `http://globalcomputing.epfl.ch/alvis`.

We believe such a situation is standard for all documents and the goal is not to compute the set $M(d)$ of relevant keys for a document d, but to identify the set $U(d)$ of all keys that have to be updated.

One way to solve this problem is to create an auxiliary indexing structure that facilitates discovery of relevant keys. Indeed, if each peer responsible for a single term could maintain a list of indexed keys containing this term, this can be used to discover relevant keys for a given document. Such a strategy would allow to discover $M(d)$ but it also has to maintain ranking information to compute $U(d)$. However, our experiments show that obtaining $U(d)$ in such a way generates significant traffic and has other drawbacks such as extra complexity, significant maintenance overhead and load imbalance.

Hence, we propose *query-driven index updates*. The main intuition behind this solution is that there is no need to update a TPL until it is requested during retrieval. Recall that full posting lists for single terms are maintained in basic index items and are used for key activations. A new TPL for an activated multi-term key is generated by applying a distributed intersection algorithm as explained in Section 5.2.3. Hence, we can reuse the same algorithm to carry out periodic TPL updates at retrieval time. Thus, indexing of a new document d only requires updates of all single-term full posting lists for all unique terms found in d. Updates of the TPLs associated with multi-term keys are done later, together with retrieval.

Whenever a multi-term key $k = t_1..t_s$ is requested during retrieval, the peers $P_{t_1}..P_{t_s}$ try to refresh the TPL $\tau(k)$ by executing the distributed intersection algorithm as explained in Section 5.2.3. There is no need to run the intersection over the full posting lists $\rho(t_1)..\rho(t_s)$ again, but only on the new portions that have been accumulated since the last TPL update. Indeed, if a set of new documents $D_{new} = d_1..d_m$ has been indexed since the last update of the TPL $\tau(k)$, the full posting lists $\rho(t_1)..\rho(t_s)$ have possibly acquired new document references from D_{new}. Thus, it is guaranteed that the intersection of only the new portions of the single-term posting lists will contain all relevant document references from D_{new} matching k. For load balancing reasons, TPL updates are done periodically and not with every retrieval request.

Figure 5.4 shows the main idea of the query-driven index update technique. Assume all single term posting lists are sorted by a timestamp. Each indexed multi-term key $k = t_1..t_s$ maintains s values $df^k_{t_1}..df^k_{t_s}$ that denote the position of the last item in the corresponding term's full posting list at the previous update (in other words df^{ab}_b denotes the b's document frequency $df_b = |\rho(b)|$ at the time of the previous update of the key ab). Obviously if $\rho(t)$ did not change since the previous update of the key k, $df^k_t = df_t = |\rho(t)|$. However if the posting lists associated with $t_1..t_s$ have changed since the previous update of k, the newly added elements are intersected and $\tau(k)$ is possibly updated. Then, $df^k_{t_1}..df^k_{t_s}$ are set to $|\rho(t_1)|..|\rho(t_s)|$ respectively.

Tracking the documents that are no longer accessible is also managed in a query-driven fashion. Failure to access a document while generating a query response could trigger removal of the corresponding document references from the posting lists that were involved in the query processing. Thus, stale entries in the index are continuously removed driven by user queries.

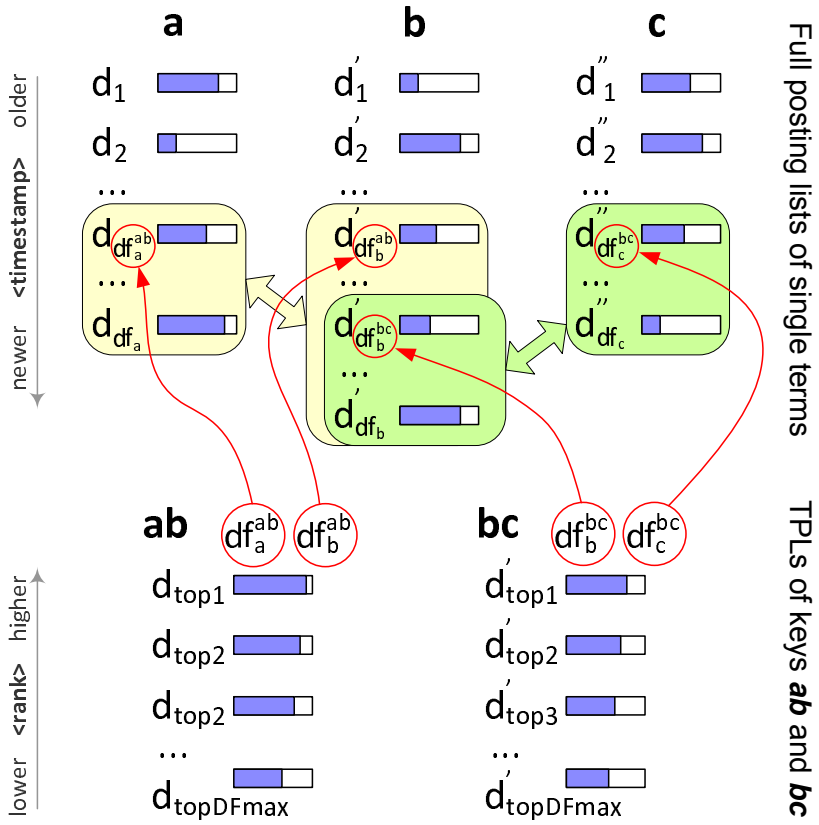

Figure 5.4. Query-driven index updates.

5.3. Indexing/Retrieval Algorithms

5.3.1. Retrieval

The pseudo code in Algorithm 5.1 defines the algorithm executed by the querying peer when it receives a new query $q = \{t_1, t_2, \ldots, t_{|q|}\}$. To process a query, the querying peer first probes all keys of size $min(s_{max}, |q|)$ extracted from the query (lines 4–26) by contacting the responsible peers. Recall that this is done by hashing a string representation of each key and following the standard DHT routing mechanism. A key of size s is probed only in case a predecessor key of size $s + 1$ have not been discovered before in the global index (line 6). For each probed key k, the function $probeKey(k)$ defined in Algorithm 5.2 is executed at the peer P_k responsible for the key k. This function tries to locate the corresponding index item in P_k's portion of the index $\mathcal{I}_{P_k}$ and to increment k's usage frequency counter. The function returns the state of the index item associated with k, which could be one of the following: $\emptyset$ (no index item), candidate or basic/active.

Algorithm 5.1 QDI query processing: $processQuery(q)$.

```
1:  result ← ∅;   /* obtained document identifiers */
2:  found ← ∅;   /* found indexed keys */
3:  candidates ← ∅;   /* keys to create new candidate items for */
4:  for i = min(s_max, |q|) downto 1 do
5:    for k ← generateNextTermCombination(q, i) do
6:      if ∀s ∈ 1..|found|, k ⊈ found[s] then
          /* route to the peer responsible for k and request its state:*/
7:        peer = DHT.route(generateKey(k));
8:        keystate = peer.probeKey(k);
          /* k is indexed ⇒ request its TPL, add to the result: */
9:        if (keystate = basic/active) then
10:         found ← found ∪ k;
11:         result ← result ∪ peer.getTPL(k);
            /* update (if needed) k's TPL: intersect new portions */
            /* of full posting lists of all single terms contained in k */
12:         if (|k| ≥ 2) and (keystate.updateRequested) then
13:           DHT.updateTPL(k, keystate.df^k_{t1..t|k|});
14:         end if
15:       end if
          /* there is no index item for k ⇒ add k to candidates: */
16:       if (keystate = ∅) and (|k| ≥ 2) then
17:         candidates ← candidates ∪ k;
18:       end if
          /* no index item for k OR k's TPL is not truncated */
          /* ⇒ delete all already found candidates that contain k: */
19:       if (keystate = ∅) or (peer.getDF(k) ≤ DF_max) then
20:         while (∃k' ∈ candidates, s.t. k ⊂ k') do
21:           candidates ← candidates \ k';
22:         end while
23:       end if
24:     end if
25:   end for
26: end for
27: DHT.createNewCandidateIndexItems(candidates);
28: return result;
```

Algorithm 5.2 QDI key probing: $probeKey(k)$.

```
1:  indexitem ← getLocalIndexItem(k); /* k's index item */
2:  if indexitem = ∅ then
3:    return ∅; /* there is no index item for k */
4:  else
5:    inc(indexitem.qf); /* increase the usage frequency counter */
6:    if indexitem.TPL = ∅ then
      /* compare the query frequency of k to the QF_min constant */
7:      if indexitem.qf ≥ QF_min then
8:        DHT.updateTPL(k); /* activate candidate indexitem */
9:      end if
10:     return candidate; /* indexitem maintains no TPL */
11:   else
12:     return basic/active; /* indexitem maintains a TPL */
13:   end if
14: end if
```

During query processing, the algorithm might discover some keys for which new candidate index items have to be created (lines 17 and 19–23). After query processing is finished, the peers responsible for such keys are contacted and requested to create the corresponding candidate index items (line 27).

Furthermore, during query processing a multi-term active index item might require a periodic TPL update (line 12). In this case the function $updateTPL(k, df^k_{t_1..t_{|k|}})$, described in Section 5.3.2 is executed (line 13), which ensures that the TPL content is up-to-date.

Finally, the usage frequency counted associated with a candidate index item can reach the QF_{min} threshold (line 5 of Algorithm 5.2) and lead to a key activation (line 8). In order to generate a new TPL for such a key, the same function $updateTPL(k, df^k_{t_1..t_{|k|}})$ is used.

5.3.2. Indexing

Whenever a new document is added to the collection, it leads to updates of the full posting lists associated with all single terms found in the document. Activating a new multi-term key (on-demand indexing) or refreshing a TPL for an active index item (index update) is done using the function $updateTPL(k, df^k_{t_1..t_{|k|}})$ during retrieval.

This function initiates the distributed intersection of the recently added portions of the full posting lists associated with all single terms belonging to the key k. The vector $df^k_{t_1..t_{|k|}} = \{df^k_{t_1}..df^k_{t_{|k|}}\}$, $\forall t_i \in k$ contains $|k|$ values corresponding to the starting positions of the document references in each full posting list of t_i that have not yet been intersected in the previous updates of the TPL $\tau(k)$ (see Figure 5.4).

For an activation of a new key (line 8, Algorithm 5.2) the $df^k_{t_1..t_{|k|}}$ vector is not specified

(filled with zeros by default) and the full posting lists of the corresponding single terms are intersected.

In this chapter we do not discuss a concrete implementation of the *updateTPL* function, however we assume that it preforms the top-k distributed intersection based on a variation of the Fagin's Threshold Algorithm (TA) [Fagin *et al.* 2001] such as Distributed Pruning Protocol (DPP) [Suel *et al.* 2003] or the Three-Phase Uniform Threshold (TPUT) algorithm [Cao and Wang 2004]. The authors of [Zhang and Suel 2005] also combined the top-k intersection with Bloom filters to further optimize the bandwidth consumption. The intuition behind these algorithms is that only top-k (top-DF_{max} in our case) document references from the resulting intersection have to be obtained, which enables efficient pruning of a large number of items associated with low scores.

To intersect posting lists $\rho(t_1)..\rho(t_{|k|})$ of $t_1..t_{|k|},\ t_i \in k$, the basic threshold algorithm scans the posting lists sorted by the document scores in parallel, and calculates the sum of scores (a threshold value) at the current position across all the lists. Each time a new document d_j is found, TA looks it up in all other lists to calculate its rank $r(d_j, k)$. TA stops when the DF_{max} documents with the rank values higher than the threshold were identified[5]. Indeed, none of the documents that have not been seen can have a score higher than the threshold, so the algorithm is always correct. Protocol optimizations in DPP [Suel *et al.* 2003] and TPUT [Cao and Wang 2004] ensure low latency and bandwidth consumption when the posting lists reside on different nodes.

Furthermore, we make an assumption that such an intersection algorithm terminates early when applied to queries with few *frequent* terms, which are typical keys in our framework. Thus, we can ensure bounded bandwidth consumption per activation at the price of a precision loss by enforcing an early termination. That is, if the intersection was not completed within a predefined traffic quota, the algorithm is forced to terminate generating a possibly incomplete TPL. However, such a TPL can still be used for query processing, possibly with a negative effect on the result quality.

5.4. Scalability

In this section we discuss the scalability of the query-driven approach in terms of bandwidth consumption. Essentially, the main reasons for scalability are the following: 1) the retrieval traffic generated while processing a query is low, since all transmitted TPLs are of bounded size, and 2) the indexing traffic needed to generate and maintain the global distributed index is manageable, as it depends on the number of indexing keys, which can be adjusted with the QF_{min} parameter. Following [Podnar *et al.* 2007], we assume that the number of documents stored per peer is bounded and thus the total number of peers N determines the maximal size

[5] Notice that the TPL update mechanism would benefit from the TA's pruning even further as the threshold value can be computed in advance based on the previous intersection results.

of the document collection.

Retrieval traffic. As answering a query leads, in the worst case, to the exploration of all the nodes in the query lattice that correspond to query term sets of at most s_{max} terms, the processing of a query requires the transmission of at most $(\log N + 1)\sum_{i=1}^{s_{max}} \binom{|q|}{i}$ messages[6]. Thus, as the size of all the transmitted messages is bounded, and, if we assume a bounded query rate for each peer in the network, the total number of transmitted messages grows with $O(N \log N)$. It corresponds to $O(\log N)$ retrieval traffic per peer, which is scalable.

Indexing traffic. The traffic generated to produce and maintain the global distributed index consists of: 1) the *single-term indexing traffic* required to populate the basic single-term index, and 2) the *query-driven indexing traffic* required to generate TPLs for newly activated keys and to update the existing ones.

Single-term indexing traffic. If we assume that the number of documents published by a peer is bounded, the number of messages that have to be transmitted in the network in order to generate the basic single-term index grows with $O(N)$. As the routing cost to deliver a message to the corresponding peer is $O(\log N)$, the total number of messages to generate the single-term index is $O(N \log N)$, which corresponds to $O(\log N)$ messages a peer has to transfer during the single-term index generation.

Query-driven indexing traffic. Each key activation triggers the on-demand indexing mechanism, which performs the distributed intersection of the corresponding single-term posting lists to generate a new TPL. While the bandwidth consumption required to generate top-DF_{max} postings stored in the TPL depends on the chosen distributed intersection algorithm, following the communication cost analysis of the Fagin's algorithms by [Suel *et al.* 2003], we can assume that the complexity of such an activation is $O(N^{\frac{s-1}{s}})$, where $s \in [2..s_{max}]$ is the number of terms in a key[7]. Furthermore, if we make an assumption that the Fagin's algorithm terminates early for queries that contain few *frequent* terms, which are typical keys in our framework – the complexity of an activation can be bounded by a constant at the price of a marginal precision loss. The overall query-driven indexing traffic thus depends on the number of actual activations.

[Podnar *et al.* 2007] showed that the number of keys generated for a given document collection by applying the HDK indexing grows linearly with the collection size. The query-driven key activation mechanism results in a substantial decrease of the number of keys as it can be viewed as an additional popularity filter based on the query distribution properties that eliminates superfluous keys. Here, we derive an upper bound on the number of keys that can be generated from the queries contained in the query log and show that it scales linearly with the query log size. Since we cannot capture analytically the correlation between the term combination popularity and its document frequency, the upper bounds are computed based on

[6] For each key it takes at most $\log N$ messages to locate the responsible peer plus one message to return the answer.

[7] Our experimental results indicate that majority of activated keys contain two terms suggest $s_{max} = 3$.

the query distribution properties only. Moreover, in Section 5.5 we will experimentally confirm the linear dependency between the number of activated keys and the size of the log, and show that despite of significant reduction of the number of indexed keys, the QDI approach causes only a marginal loss of the answer quality even for Web-size document collections and real query distributions.

If we abandon the correlation between the term combination popularity and its posting list size, the number of possible keys selected for indexing depends only on the query log properties. We make the assumption, backed by preliminary empirical analysis (see Figure 5.14), that the number of multi-term combinations found in the query log follows the Pareto [Reed 2001] distribution. We use this assumption to analyze the number of keys that can be potentially activated. This number, however, is an upper bound and is significantly reduced in practice by applying size and redundancy filters.

We are interested in deriving the number of multi-term keys with minimum usage frequency (popularity) QF_{min} after $|L|$ queries were observed in the query log. Recall, that the number of single term keys does not depend on the query distribution as all terms found in the document collection are indexed. According to Heaps' law [Heaps 1978], the number of distinct terms grows as $O(\sqrt{|\mathcal{D}|})$ with the size of the document collection $|\mathcal{D}|$, and, therefore can be practically managed in a P2P network. For our future analysis, we ignore single term keys and concentrate on combinations of size 2 and larger.

We assume that usage frequencies of keys of sizes $2, .., s_{max}$ are Pareto distributed, *i.e.*, the probability that the usage frequency qf_k of a key k is higher or equal to a predefined QF_{min} is given by:

$$P(qf_k \geq QF_{min}) = (QF_{min})^{-\alpha},\ |k| \geq 2,$$

where $\alpha > 0$ denotes the Pareto index.

We define $|\mathcal{K}_1|$ as the number of multi-term keys that have appeared only once in the query log. The total number of multi-term keys γ found in a query log L is thus

$$\gamma = |\mathcal{K}_1| \int_1^{+\infty} (QF_{min})^{-\alpha}\ d(QF_{min}).$$

Assuming[8] $\alpha > 1$:

$$\gamma = \frac{|\mathcal{K}_1|}{\alpha - 1}$$

or

$$|\mathcal{K}_1| = \gamma\ (\alpha - 1).$$

The number of multi-term keys to be activated for the chosen QF_{min} is the following:

$$|\mathcal{K}_{QF_{min}}| = |\mathcal{K}_1|\ P(qf \geq QF_{min}) =$$

$$= (\alpha - 1)\ (QF_{min})^{-\alpha}\ \gamma = O(\gamma).$$

[8] Our experiments suggest values of α around 1.8, see Figure 5.14.

Assuming the number of multi-term keys extracted from a query is bounded by a constant, calculated from the maximum query size, we can write:

$$|\mathcal{K}_{QF_{min}}| = O(|L|).$$

Therefore, the number of activated keys grows linearly with the number of queries submitted to the system. Moreover, increasing the QF_{min} parameter results in decreasing the number of activated keys according to the power law. We also justified this result experimentally, see Figure 5.14.

Thus, we showed that the activation rate linearly depends on the global query rate. If we assume that the query traffic per peer is bounded, the activation rate grows with $O(N)$. Hence, by enforcing a constant complexity of an activation we achieve a scalable traffic in the P2P setup.

In addition, the traffic can be further controlled at the price of retrieval quality degradation by tuning the QF_{min} parameter for a given time interval. Our extensive experimental evaluation (see Section 5.5) shows that the retrieval quality remains acceptable even for Web-scale document collections with reasonably chosen time interval and QF_{min} values.

Furthermore, in practice, we utilize a novel DHT-level congestion mechanism [Klemm *et al.* 2006], which optimizes the usage of the underlying DHT network resources. This module takes care of the on-the-fly aggregation of the messages with the same next-hop destination and facilitates efficient P2P network utilization at peak loads.

5.5. Experiments

We showed that the QDI approach scales to Web sizes with respect to the size of the index. However, by pruning superfluous keys we have to tolerate a certain degradation of the retrieval quality. Since we cannot capture analytically the correlation between the popularity of a certain term combination and its document frequency, we conducted several large-scale experiments (Sections 5.5.1 and 5.5.2) with real query-logs to investigate the retrieval performance of our query-driven approach.

In Section 5.5.3 we also confirm that retrieval precision of our approach achieved for a random set of real queries is fully comparable to the one obtained with a state-of-the-art centralized query engine.

The second set of experiments analyzing the performance of the QDI approach in a dynamic setting and investigating the size of the query-driven index is presented in Sections 5.5.4 and 5.5.5. We report a significant reduction of the index size compared to the HDK-based index, which in turn yields a significant bandwidth consumption decrease when compared to the HDK approach.

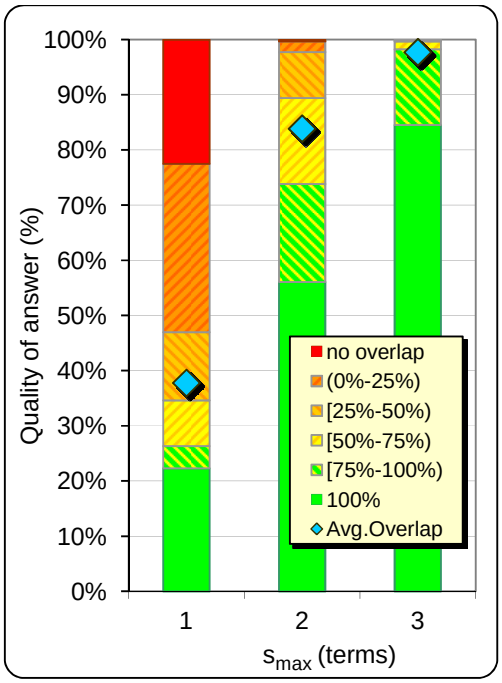

Figure 5.5. Google experiment: maximal overlap achieved for different values of s_{max} (assuming *all* possible combinations of size $1..s_{max}$ are indexed).

5.5.1. Retrieval Quality Experiments with the AOL Query-Log

To analyze retrieval performance, we conducted several experiments using query logs from the AOL search engine. These logs contain more than $17M$ of real queries collected from $650K$ users during 3 months, from March to May 2006. We discarded the information about user sessions and obtained a large list of queries sorted by timestamps. In addition, we considered only unique entries in each user session, so the repetition of a query by the same user does not affect the query popularity distribution. Finally, we filtered out queries corresponding to Web site URLs, which represent a large fraction (about a third) of all queries, but are not interesting for our experiments[9].

The aim of this set of experiments was to evaluate the impact of query-driven indexing on retrieval quality in the context of real life Web-scale document retrieval. To do so, we randomly generated a test set of $2,000$ queries taken from the last day of the AOL log. We then crawled Google's top-20 results for each query in the test set. In the following, we will refer to these top-20 results for a query q as the *reference result* for q.

For each test query, we removed the common stop words, applied the stemmer [Porter 1980], sorted the terms in alphabetical order[10] and generated all possible term combinations. Then, for each of these combinations, we computed their query frequencies using the AOL

[9] Such queries can be easily resolved in our framework by treating URLs as single terms.
[10] The sorting is performed to diminish the effect of term ordering on Google's ranking.

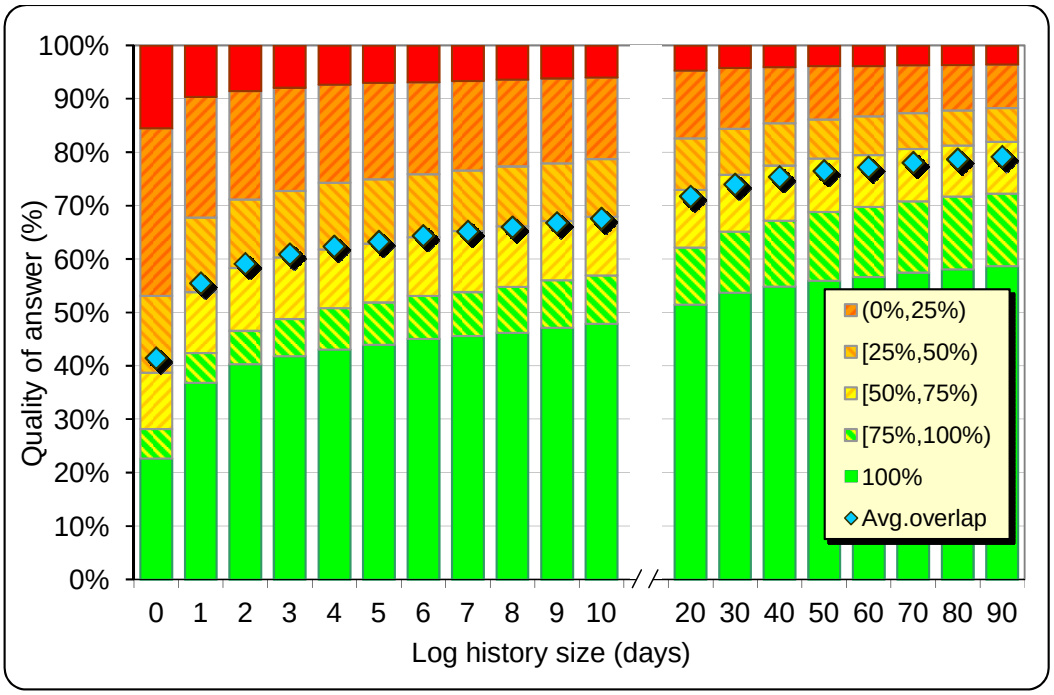

Figure 5.6. Google experiment: overlap achieved for different sizes of the query log measured in days ($QF_{min} = 1,\ DF_{max} = 600,\ s_{max} = 3$).

query log for the previous 3 months and crawled Google's top-DF_{max} results for all generated term combinations. In our query-driven index, these results would be contained in the TPLs associated with indexed term combinations, provided that each peer uses the same ranking mechanism as Google.

To evaluate the retrieval quality for a query q, we measure the *overlap* between the reference top-20 results for q and the union of the TPLs associated with the term combinations (keys) that are: 1) contained in q, and 2) indexed. The overlap is expressed as the fraction of the reference top-20 result that appear in the generated union. In other words, the overlap corresponds to precision@k when Google considered as the reference.

In all experiments, we categorize the overlap values into the following categories:

1. $overlap = 0\%$, *i.e.*, no result from the reference top-20 appears in the union,
2. $overlap = (0\% - 25\%)$,
3. $overlap = [25\% - 50\%)$,
4. $overlap = [50\% - 75\%)$,
5. $overlap = [75\% - 100\%)$ and
6. $overlap = 100\%$, *i.e.*, all results from the reference top-20 appear in the union.

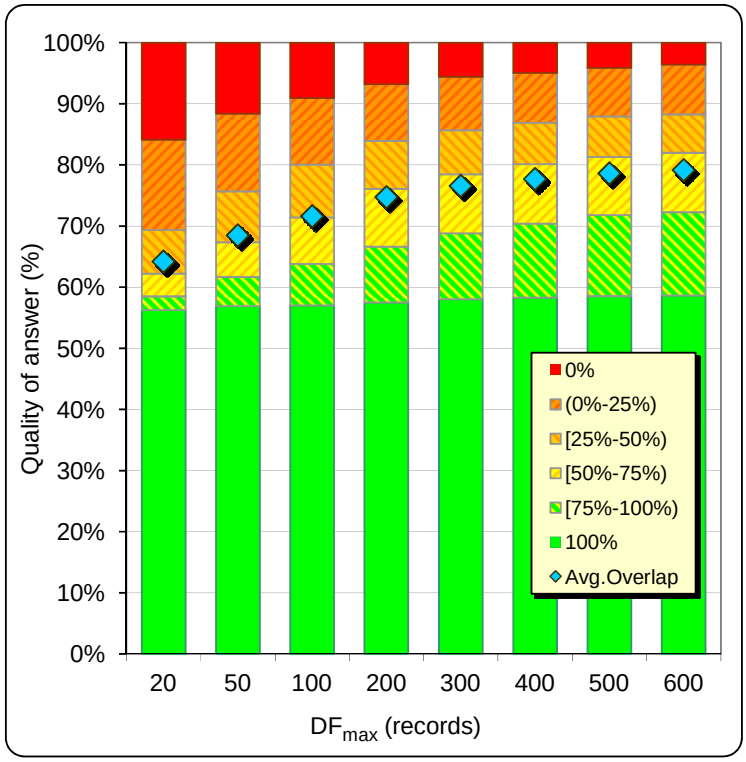

Figure 5.7. Google experiment: overlap achieved for different values of DF_{max} (QF_{min} =1/3 months, $s_{max} = 3$).

Impact of s_{max}. Figure 5.5 shows the achieved overlap if *all* possible combinations of size $1, .., s_{max}$ are indexed for different values of s_{max}. From the figure, considering term combinations with 1, 2 or 3 terms is sufficient to achieve overlap values as high as 97%. Thus, in our experiments we fix $s_{max} = 3$.

Impact of the log size. Figure 5.6 shows the achieved overlap as the function of the size of the query log, measured in days. QF_{min} was set to 1, *i.e.*, a term combination should be encountered at least once in the previous query history to be activated and indexed. We set the DF_{max} parameter to 600 and s_{max} to 3.

With this setting, one can see that, starting from the poor performance of the single term index[11], the overlap rapidly grows with 100-200K queries being processed per day. The three months query log yields an overlap of about 80%. Notice that our approach could use larger query logs, which would further improve the retrieval quality. It is also important to mention that for more than 80% of the test queries at least 10 out of 20 reference results were found, whereas a very low fraction (about 3-4% of the queries) performed poorly returning no reference result. We analyzed queries with poor overlap and identified that in 30-40% of the cases the queries were misspelled. Thus, if they are treated properly, the overlap values can be increased.

[11] Recall that if no query log is available, only single term keys are indexed by default.

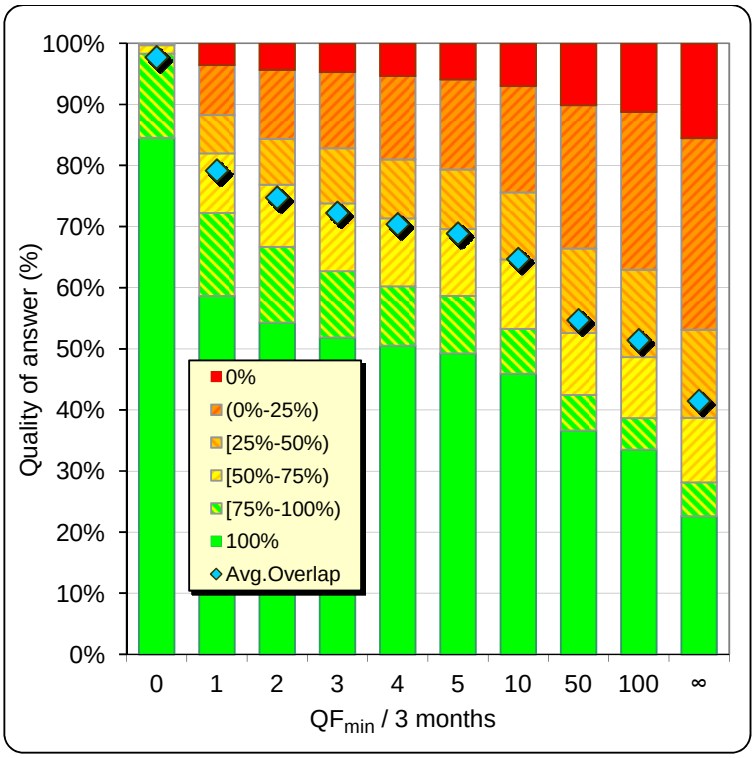

Figure 5.8. Google experiment: overlap achieved for different values of QF_{min}/3 months ($DF_{max} = 600,\ s_{max} = 3$).

From Figure 5.6, we can conclude that taking popular combinations: 1) significantly increases retrieval quality when compared to the single term index (a twice higher overlap with the 90-days query log), and 2) yields an overall satisfactory retrieval quality.

Impact of DF_{max}. We used the 3-month query log, set QF_{min} to 1 and investigated the impact of DF_{max} on retrieval quality. Figure 5.7 shows the achieved overlap for different values of DF_{max}. From the figure, even small DF_{max} values of 200–500 are sufficient to achieve good retrieval quality. It is also interesting to observe that changing DF_{max} hardly affects the fraction of queries with the 100% overlap, with a growth from 56% to 59% for DF_{max} changing from 20 to 600 respectively. The DF_{max} value however affects the average overlap.

Impact of QF_{min}. Finally, Figure 5.8 shows the decrease in retrieval quality when increasing QF_{min} from 0 (all possible combinations are indexed) up to infinity (basic single term index). From the figure, query-driven indexing of multi-term keys has the potential to double the overlap compared to the basic single-term index (overlap=80% with $QF_{min} = 1$ compared to 41% with $QF_{min} = \infty$). As before, s_{max} was set to 3 and DF_{max} to 600.

Based on these results, we can assume that, in practice, the QF_{min} parameter should be chosen in the 5-20 range resulting in a 60%-70% overlap with our settings. In this case, the period during which we keep the query statistics should be increased accordingly.

5.5.2. Retrieval Quality Experiments with the Wikipedia Query-Log

We also used the Wikipedia query log introduced in Section 4.4. Recall that it contains more than $9M$ queries issued to the Wikipedia on-line encyclopedia during September and October 2004. We generated a test set of $3,000$ queries and, for each of these queries, built a reference result by retrieving the top-20 results produced by the *Google* and *Yahoo!* search engines.

We extracted all the term combinations present in the query log and evaluated their popularity. The query log contained more than $9M$ queries and we observed around $10M$ unique term combinations. Since users usually search for a concrete article in the Wikipedia encyclopedia, the popularity distribution of the Wikipedia query-log is much more skewed than of a query-log of a typical Web search engine (*e.g.*, the AOL query-log).

Figure 5.9 shows the average overlap between the result we get from our system and the reference set from the *Google* and *Yahoo!* search engines in the same way we did in Section 5.5.1. First observation we can make from the figure is that due to the fact that the Wikipedia query-log is very skewed, the average overlap is very high, especially given that the DF_{max} parameter was set to 100. Second, it confirms that setting $s_{max} = 3$ is a good design choice – the line that corresponds to $s_{max} = \infty$ is not depicted on the plot just because it coincides with the line for $s_{max} = 3$. Most importantly, it shows that the overlap values are very similar between *Google* and *Yahoo!*, which means that our solution is quite robust to reasonable variations in the scoring function.

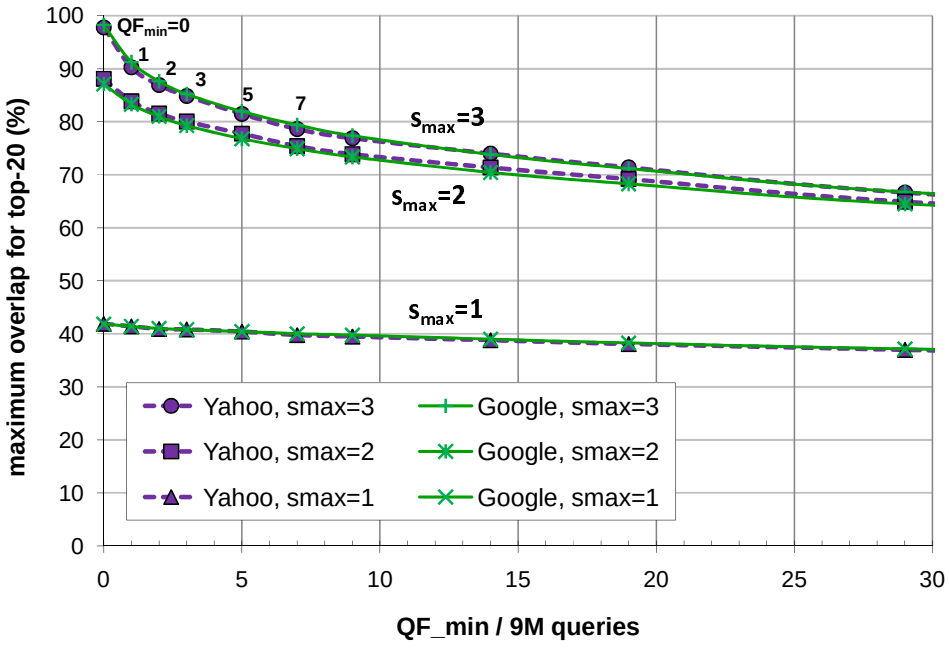

Figure 5.9. The average overlap obtained with the *Google* and *Yahoo!* search engines.

Precision	$DF_{max} = 100$				$DF_{max} = 500$				ST-BM25
	QF_{min}=∞	QF_{min}=5	QF_{min}=3	QF_{min}=1	QF_{min}=∞	QF_{min}=5	QF_{min}=3	QF_{min}=1	
P@5	0.306	0.345	**0.347**	0.341	0.345	0.343	0.343	0.343	0.337
P@10	0.266	0.299	0.295	0.294	**0.307**	0.302	0.303	0.302	0.298
P@15	0.237	0.267	0.267	0.267	0.276	0.279	**0.280**	0.278	0.278
P@20	0.212	0.243	0.243	0.246	0.254	**0.259**	**0.259**	**0.259**	0.257
P@30	0.174	0.206	0.209	0.212	0.214	0.221	0.221	0.224	**0.226**
P@50	0.139	0.169	0.171	0.174	0.175	0.181	0.181	0.183	**0.186**
P@100	0.097	0.126	0.127	0.130	0.128	0.135	0.135	0.136	**0.140**
#docRef	236.7	184.6	179.1	173.3	1148.2	880.9	846.2	813.2	193652.4

Table 5.3. QDI: Precision@k for the TREC experiment.

5.5.3. TREC Experiment

To further evaluate the retrieval quality of our approach, we also used the WT10G collection[12] that contains 1'692'096 documents. 100 test queries were selected from the Ad hoc topics of the Web Track in TREC-9 and 10[13]. We processed title-only queries because queries with additional fields were not used in the real Web search. The standard TREC assessments supplied by the U.S. National Institute of Standards and Technology were used. We used the Terrier[14] engine with the BM25 weighting scheme to compute top-DF_{max} documents stored in the TPLs, and also to compute the final ranked results.

After processing 17M queries from the AOL query log to generate the query-driven index, we submitted the 100 TREC queries to the system. Then we compared our results to the ones returned by the centralized Terrier engine (we denote this by ST-BM25, i.e., single term indexing using the BM25 weighting scheme). DF_{max} was set to 100 and 500. Notice that the DF_{max} parameter is useful to control the trade-off between the retrieval cost and the retrieval quality (the smaller the DF_{max}, the lower the bandwidth consumption during retrieval). QF_{min} was set to 1, 3, 5 and ∞, where $QF_{min} = \infty$ means that no key is activated and only the basic single term index is used to process the queries.

Table 5.3 shows the achieved precisions at K (P@K). The highest value in each line of the table is highlighted in bold. In general, the results achieved by our system (excluding $QF_{min} = \infty$, $DF_{max} = 100$) are slightly better than ST-BM25 for K≤20. For K>20, our system starts loosing some relevant documents, when compared to ST-BM25, because we only store at most top DF_{max} document references per key. However, we believe this should not be a problem in the context of Web search, where users are usually only interested in the top 10-20 documents.

In addition, for K>20, Table 5.3 also shows that, with a higher value for DF_{max}, our system is becoming similar to ST-BM25 (in fact, if $DF_{max} = |\mathcal{D}|$, our system is equivalent

[12] `http://ir.dcs.gla.ac.uk/test_collections/wt10g.html`
[13] TREC Web Track, `http://trec.nist.gov/data/webmain.html`
[14] Terrier search engine, `http://ir.dcs.gla.ac.uk/terrier/`

to ST-BM25). In the worst case, when $DF_{max} = 100$ (we only keep top-100 documents in the posting lists) and $QF_{min} = \infty$ (the query driven mechanism is not applied), our system retrieves 75% of the relevant documents retrieved by ST-BM25 at top-50 (0.139/0.186) and 89% at top-10 (0.266/0.298). Notice that these values are already quite high due to the relatively small size of the WT10G collection. In general, the query-driven technique with reasonable values of QF_{min} performs similarly to ST-BM25.

The last line in the table shows the average number of transmitted document references during the processing of the 100 TREC queries, which indicates the bandwidth consumption during retrieval. For ST-BM25, we simulate the naïve approach where the full posting lists are transmitted to the querying peer for each of the terms in the query. Obviously, with smaller values of DF_{max}, we achieve lower bandwidth consumption. Since our posting lists are truncated to a constant size, the bandwidth consumption will remain constant when the size of the collection increases, as shown by [Podnar *et al.* 2007].

Finally, the TREC experiment confirms the conclusion that the query driven indexing approach indeed delivers a retrieval quality that is fully comparable to the one of a centralized single-term index, and, at the same time, guarantees scalable traffic during retrieval.

5.5.4. P2P Index Simulations

In this section we analyze the performance of our approach in a dynamic setting. Starting from the basic single-term index, we observe how processing of new queries triggers indexing of newly activated keys and improves the retrieval quality. We conducted a set of experiments with a *small* Wikipedia document collection containing 650K articles and the Wikipedia query log introduced in Section 4.4. The experiments were carried out with the following parameters: $DF_{max} = 100$, $QF_{min} = \frac{4}{2M}$ (*i.e.*, a key is activated if it occurs at least 4 times among the 2M recent queries) and $s_{max} = 3$. Notice, that DF_{max} is smaller and QF_{min} is larger compared to the values we used in Section 5.5.1 because the Wikipedia collection is much smaller than the set of pages indexed by *Google*.

Figure 5.10 shows the number of generated keys as queries are being processed. Notice that out of $1.3M$ single-term keys, less then $200K$ were actually used in queries, and that the total number of generated keys was reduced by $1-2$ orders of magnitude when compared to the HDK approach without the query-driven key activation. We estimated that the HDK approach would produce around $65M$ keys in this scenario, whereas our approach requires only $0.5M$ keys to be activated with $QF_{min} = 2/4M$, in addition to the $1.3M$ single-term keys, while, as shown in Figure 5.11, the retrieval quality remains reasonable.

We also measured the average overlap for the query results obtained with our approach compared to the full single-term index based on the Terrier retrieval engine[15]. In this experiment we measure the overlap between our approach and the results obtained for the full

[15] `http://ir.dcs.gla.ac.uk/terrier`

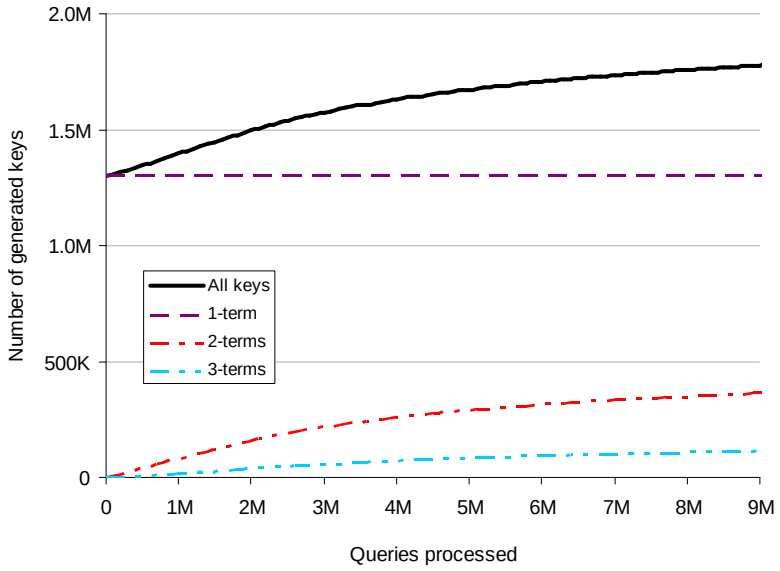

Figure 5.10. Number of generated indexing keys depending on the number of processed queries.

queries using the Terrier search engine over the Wikipedia collection. The obtained overlap values are shown in Figure 5.11. These values show that the retrieval quality grows quite fast with the number of processed queries, starting from a relatively low value corresponding to the single term index. At each point the overlap value was obtained by processing of a test set of 50K queries with a given state of the index. Index updates were frozen during the overlap measurements. For comparison we show similar plot obtained for $QF_{min} = \frac{3}{2M}$.

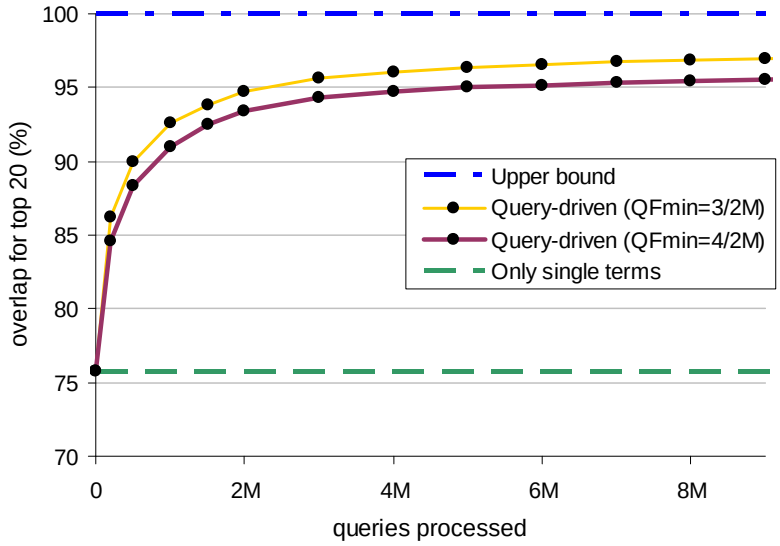

Figure 5.11. Average overlap depending on the number of processed queries.

Finally, Figure 5.12 shows the upper bounds for the overlap that can be achieved with our indexing mechanism for different QF_{min} values. This figure is similar to Figure 5.9, but, as the document collection is much smaller shows higher overlaps.

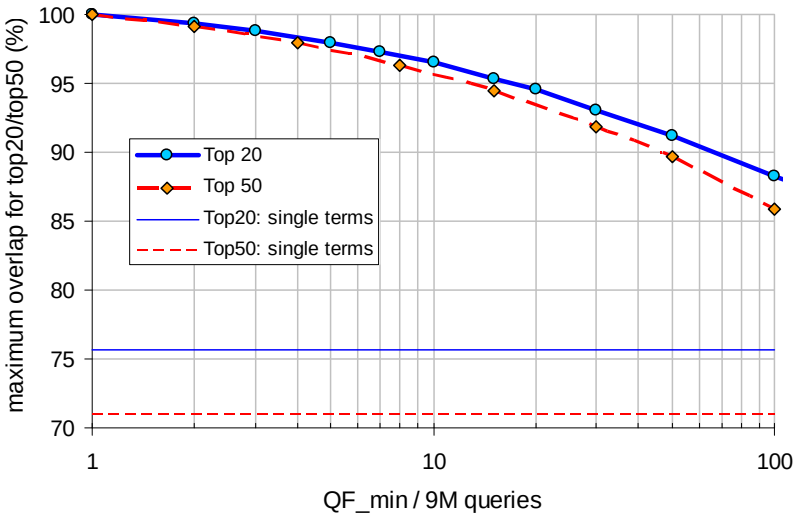

Figure 5.12. Overlap upper bound for different values of QF_{min} with the small Wikipedia document collection.

The plots show the obvious tradeoff between the quality of answers and the number of generated keys (and, therefore, the amount of the indexing traffic) in the network. Finally, we showed that real savings in storage and bandwidth requirements can be achieved with a marginal degradation of the answering quality.

5.5.5. Experiments Investigating the Index Size

In this section we analyze the number of keys present in the query driven index and compare it to the HDK approach. We simulated the query-driven retrieval mechanism and analyzed the number of activated keys of size 2 and 3 for the AOL query log. We plot in Figure 5.13 the number of keys being activated when processing new queries for different threshold values of QF_{min}[16]. As we proved in Section 5.4, this number grows linearly with the size of the query log $|L|$ and the QF_{min} parameter can be used to adjust the slope. Figure 5.14 also confirms that the number of multi-term combinations in the query log follows the Pareto distribution. Thus, the number of keys decreases according to the power law with the increase of the QF_{min} parameter.

Recall that the total number of indexed keys is composed by all single term keys found in the document collection and all multi-term keys activated during retrieval. According to Heaps' law [Heaps 1978], the number of distinct terms grows as $O(\sqrt{|\mathcal{D}|})$ with the size of the document collection $|\mathcal{D}|$. Additionally, the number of activated keys grows linearly with the size of the query log $|L|$. On the other hand, the size of the index for the HDK approach grows linearly with the number of documents $|\mathcal{D}|$ and does not depend on the query log. We made a speculative comparison between the number of keys stored in the index for both HDK

[16] We considered the first 45 days of the log only due to the simulator's memory restrictions.

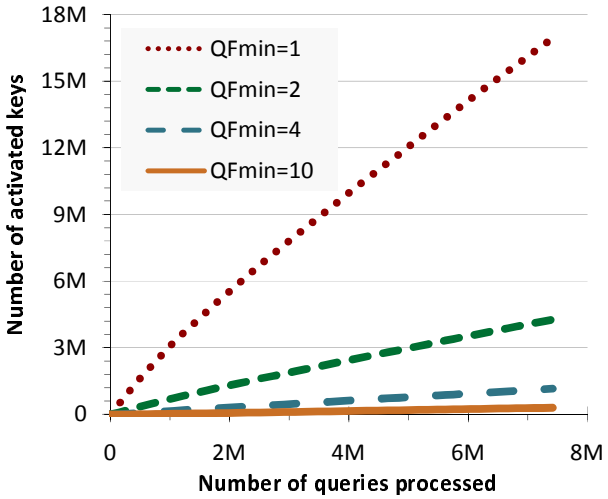

Figure 5.13. The number of activated keys in the query-driven index with new queries being processed.

and QDI approaches. Figure 5.15 shows how the total number of keys grows with the size of the document collection for both approaches (we varied the size of the query-log $|L|$ and set $QF_{min} = 2$ to compute the curves for the QDI approach). It is easy to see that due to the fact that the size of the query log is potentially much smaller than the size of the document collection, the query-driven approach would use orders of magnitude less resources than the HDK approach.

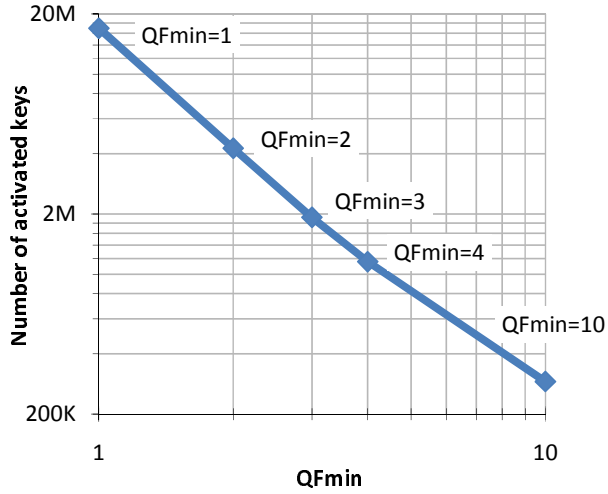

Figure 5.14. The number of activated keys in the query-driven index for different values of QF_{min} after processing 7.5M queries (first 45 days of the AOL query log).

In [Skobeltsyn *et al.* 2007b] we also provided initial comparisons of the bandwidth consumption during indexing, assuming for the QDI approach that each key activation is resolved

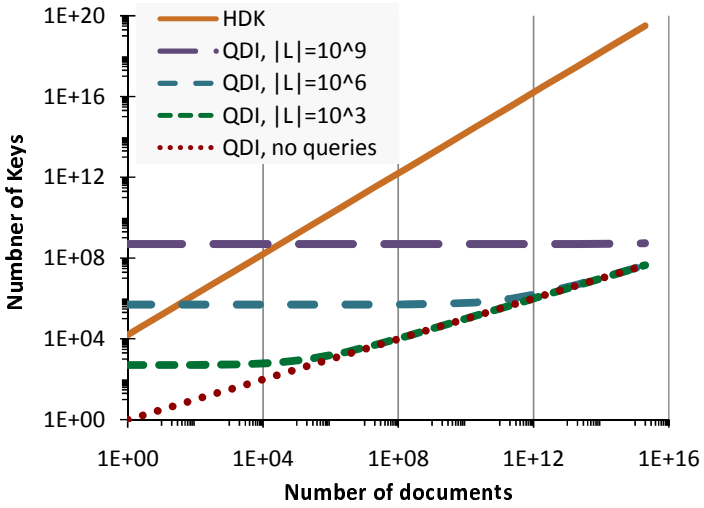

Figure 5.15. Speculative comparison of the total number of keys stored in the index for the HDK and QDI approaches.

using a broadcast. We showed that in realistic scenarios the query-driven index requires several orders of magnitude less bandwidth than the HDK approach. Employing the distributed intersection algorithm instead of broadcasting each activation would sufficiently reduce the traffic requirements, but might increase the risk of load imbalance.

5.6. AlvisP2P Prototype*

In this section we briefly describe a prototype of the *AlvisP2P* IR engine [Luu 2007; Luu *et al.* 2006, 2008], which implements efficient retrieval with multi-keyword queries from a global document collection available in a P2P network using either the HDK or QDI approach. While operational P2P-IR systems are being deployed, such as Faroo[17] or YaCy[18], AlvisP2P prototype is more a research platform that permits us to test sophisticated indexing schemas such as HDK and QDI.

5.6.1. AlvisP2P Architecture

The AlvisP2P architecture is layered in order to separate different conceptual levels, and allow the higher layers to use the functionalities provided by the lower ones. Altogether, it comprises the following layers:

* AlvisP2P implementation is a joint effort of a large team of people including Toan Luu (the main author and the coordinator of the project), Fabius Klemm (P2P layer), Maroje Puh (ranking), Gleb Skobeltsyn (QDI) and others. AlvisP2P Web-site: `http://globalcomputing.epfl.ch/alvis`.

[17] `http://www.faroo.com`

[18] `http://yacy.net`

L1 A *transport layer*, which provides the means for direct communication between two peers;

L2 A *peer-to-peer layer*, which maintains the Peer-to-Peer overlay infrastructure;

L3 A *distributed IR layer*, which provides the basic functionalities related to document management, in particular the ones related to distributed IR;

L4 A *ranking layer*, which implements functionalities related to distributed document ranking; and

L5 A *local search layer*, which implements possibly sophisticated local IR models.

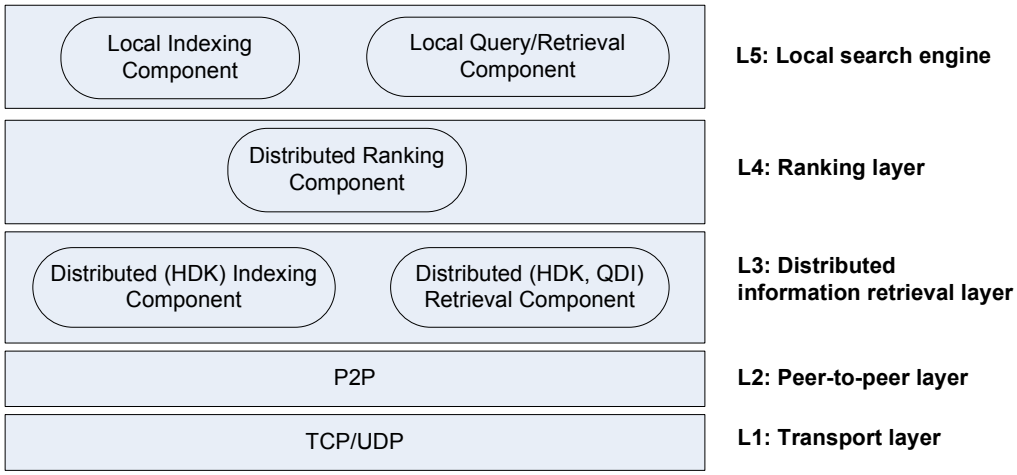

Figure 5.16. AlvisP2P architecture – layered view.

Figure 5.16 shows all major AlvisP2P functionalities positioned in the corresponding layers. While components in higher layers exclusively rely on the functionalities provided by lower layers, the architecture does not prevent from having different types of peers integrating in a more or less extensive way the layers from 3 above. For example, a lightweight peer could only integrate layers 1 to 4, while a peer associated with a more sophisticated local search engine could exploit all 5 layers. The discussion on the performance issues of such a system is presented by [Luu *et al.* 2006].

Layers 1 and 2 implement the peer-to-peer overlay infrastructure. Layer 2 (or P2P layer) consists of a Distributed Hash Table (DHT) that is able to sustain high traffic loads. Peers build routing tables of size $O(\log N)$, which results in an expected routing cost of $O(\log N)$ hops (where N is the number of peers in the network). As it uses the concept of "hop space" for routing table construction, the DHT supports arbitrary skews in the distribution of the peers in the identifier space [Klemm *et al.* 2007]. In addition, we integrated a congestion control mechanism into our DHT [Klemm *et al.* 2006] to efficiently handle the large amounts

of messages generated by the information retrieval application and to prevent the DHT from congestion collapses.

Layer 3 provides the features related to distributed information retrieval and implements one of the aforementioned techniques, *i.e.*, indexing with highly discriminative keys (HDK) or query driven indexing (QDI). This layer deals with the task of *key-based indexing*, *i.e.*, finding the set of keys and associated posting lists for a given document, and the *querying* task, *i.e.*, given a query, finding corresponding keys in the global P2P index, retrieving the postings associated with those keys and merging the result set for ranking. Additionally, the QDI approach uses Layer 3 to collect the popularity statistics that define the keys to be indexed.

Layer 4 is responsible for ranking. Depending on the ranking model[19], it might use global document frequencies, average document length, term frequencies and other statistical information, which are stored in the P2P network, to compute the relevant scores of documents w.r.t the query.

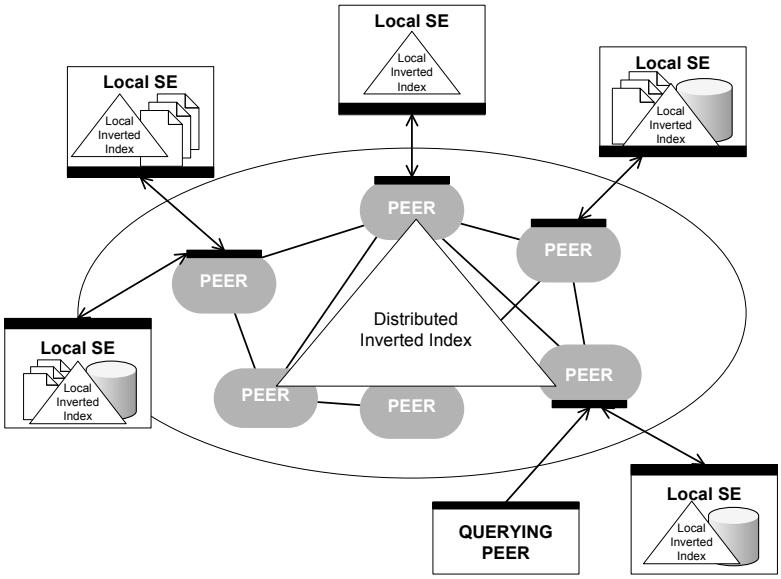

Figure 5.17. AlvisP2P network.

Layer 5 implements a possibly sophisticated "local search engine". For example, as shown in Figure 5.17, such a search engine can use specialized document processing for its local collection to build semantically rich indexes enhanced by various ranking strategies[20]. The local search engine interacts with the associated peer through a generic API and uses a well-defined communication protocol to submit the index of its local collection to the global P2P network and to process queries.

More precisely, the answer to a given query can be:

[19] Currently, the prototype it using the state-of-the-art BM25 ranking function. Notice, however, that any other function could be used instead, provided that the required global statistics are available in the P2P network.

[20] *E.g.*, it can support complex structured queries or/and employ a particular ranking strategy.

- either produced exclusively using the information available in the distributed index and a uniform distributed ranking model; in this case the retrieval mechanism guarantees good response times, but, possibly, at the price of a lower precision;
- or refined in a second step during which the query is forwarded to the local search engines associated with the peers holding the documents found in the first step; in this case the retrieval might be slower (as it requires several interactions), but can benefit from the advanced features made available by the local engines.

5.6.2. AlvisP2P Client Software

Joining an AlvisP2P network is as simple as downloading and installing the peer client software. The user only has to specify few communication parameters, such as the IP address of a contact peer and the communication port. The client software includes a Web server that can be accessed by anyone through a Web browser to query the AlvisP2P network. Alternatively, the default standalone client can be used, which allows only the local user to access the AlvisP2P network from this peer. Figure 5.18 shows the interface of the AlvisP2P client.

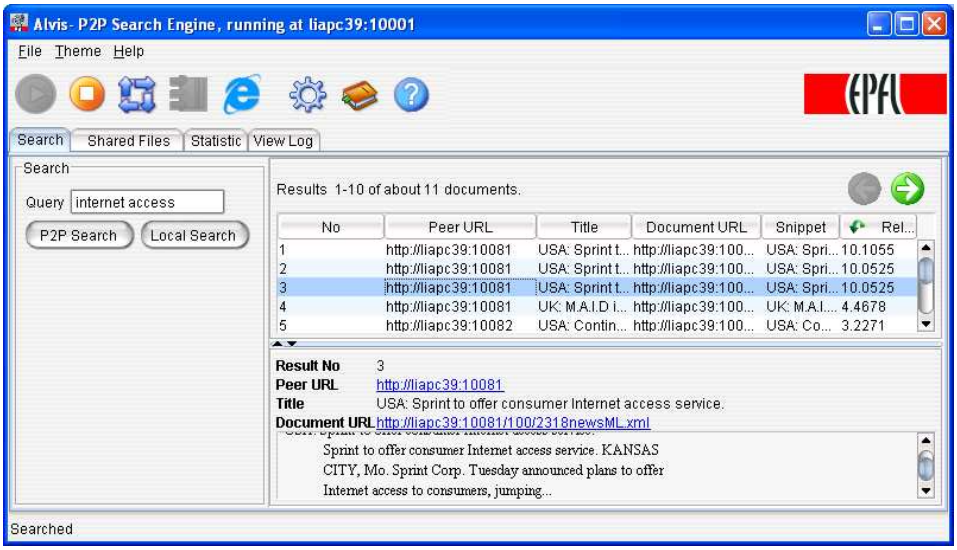

Figure 5.18. Screenshot of the AlvisP2P client software.

To make documents searchable by other peers in the network, the user simply puts them in the shared directory of his/her peer and uses the AlvisP2P software to index them. The following document types are supported: *txt*, *xml*, *html*, *doc*, *pdf* or xml-based Alvis format. The index of a local shared document collection is implemented using the Terrier search engine[21].

As local documents always remain at the peer that holds them, the document owner can define specific access rights for them. For example, the user can choose that a document can be freely accessible or has a limited access controlled by a username and a password.

[21] `http://ir.dcs.gla.ac.uk/terrier/`

The AlvisP2P prototype is available at: `http://globalcomputing.epfl.ch/alvis`.

5.7. Conclusions

Using a structured P2P network for distributing the load among a large number of interconnected nodes represents a promising approach for indexing very large document collections, but poses serious challenges on the design of the distributed index in order to remain scalable with respect to bandwidth consumption, storage space and load balancing at indexing and retrieval.

In this chapter we described a novel query-driven indexing strategy based on indexing of carefully selected and popular term combinations that guarantees scalable storage and bandwidth requirements. We also proposed a query-driven update mechanism that facilitates keeping the index up-to-date with document collection changes. Our approach is shown to be viable for Web-scale document collections based on the experimental evaluation of the retrieval performance for Web-size document collections and query logs.

We showed that using such a highly distributed cache based on the truncated posting lists for popular term combinations is beneficial for large-scale P2P information retrieval. Our further analysis revealed that trivial porting of this solution for the search engine architecture is not beneficial, mainly because a centralized results caching component can be used instead. Clearly, most of the queries that return good quality results in our system would also be hits in the centralized results cache.

Comparison with a centralized results cache. To estimate whether our query-driven indexing technique simply substitutes a standard results cache, we computed the theoretically maximal hit rate[22] of the results cache with the AOL query log used in the experiments. We obtained the maximal hit rate just below 41%. We can compare this value to up to 60% of queries returning all top-20 correct results and up to 80% returning at least half of them with the query-driven indexing. This shows that a substantial fraction of queries that are misses in the (centralized) results cache could still be successfully processed with our system. Furthermore, it suggests that with more skewed query logs (such as the one used in Chapter 6) our results can be even better.

The observation above triggers another interesting question: How the changes in the query stream caused by the results cache affect the performance of query processing in modern Web search engines? We address this question in the next chapter.

[22] We used the formula $hits_{max} = (1 - \frac{distinct\_queries}{all\_queries})$, which measures the maximal hit rate of an indefinite-capacity cache.

Chapter 6

ResIn: A Combination of Result Caching and Index Pruning for High-Performance Web Search Engines*

6.1. Introduction

In this chapter we switch our attention from the Peer-to-Peer index organization to the classical Web search engine (WSE) architecture with the purpose of investigating advantages of query-driven indexing in this setting. In particular, we explore the results caching and index pruning techniques that employ the information about the past queries for query processing optimization.

Major Web search engines process thousands of queries per second[1] over billions of indexed Web pages[2] with sub-second response times. To sustain such a load, they rely upon large complex systems using thousands of servers, interconnected through different networks, and often spanning multiple data centers. Servers in these systems are often grouped according to some functionality (*e.g.*, front-end servers, back-end servers, brokers), and requiring that the groups of servers interact increases the amount of time and resources needed to process each query. It is therefore crucial for high-performance Web search engines to design mechanisms that enable a reduction of the amount of time and computational resources involved in query processing to support high query rates and ensure low latencies.

* The material presented of this chapter was published in the proceedings of the 31st International ACM SIGIR Conference (SIGIR'08) [Skobeltsyn *et al.* 2008]. This work was done during Gleb Skobeltsyn's internship visit at Yahoo! Research, Barcelona.

[1] `http://www.comscore.com/press/release.asp?press=2230`

[2] `http://www.google.com/press/funfacts.html`

Results caching is an efficient technique for reducing the query processing load, hence it is commonly used in real search engines. This technique, however, bounds the maximum hit rate due to the large fraction of singleton queries, which is an important limitation. In this chapter we propose *ResIn* – an architecture that uses a combination of *res*ults caching and *in*dex pruning to overcome this limitation. We argue that results caching is an inexpensive and efficient way to reduce the query processing load and show that it is cheaper to implement compared to a pruned index. At the same time, we show that the index pruning performance is fundamentally affected by the changes in the query traffic that the results cache induces. We experiment with real query logs and a large document collection, and show that the combination of both techniques enables efficient reduction of the query processing costs and thus is practical to use in Web search engines.

Typically, search engines use caching to store previously computed query results, or to speed up the access to posting lists of popular terms [Baeza-Yates *et al.* 2007b]. Results caching is an attractive technique because there are efficient implementations, it enables good hit rates, and it can process queries in constant time. Moreover, as the query results are available after a query is processed by the index, having a results cache is simply a matter of adding more memory to the system to hold such results temporarily.

One important issue with caches of query results is that their hit rates are bounded. Due to the large fraction of infrequent and singleton queries, even very large caches cannot achieve hit rates beyond $50 - 70\%$, independently of the cache capacity. To overcome this limitation, a system can make use of posting list caching or/and employ a pruned version of the index, which is typically much smaller than the full index and therefore requires fewer resources to be implemented. Without affecting the quality of query results, such a static pruned index is capable of processing a certain fraction of queries thus further decreasing the query rate that reaches the main index, as for example shown by [Ntoulas and Cho 2007].

We show that there are several benefits of using results caching and index pruning together, including higher hit rates and reduced resource utilization. Although index pruning has been studied in the literature, to our knowledge, we are the first to present results on the impact of results caching on index pruning in an architecture that is highly relevant for practical Web search systems. In such an architecture, the stream of queries that has to be processed by the pruned index differs significantly from the the original query stream because many queries are filtered out by the results cache. This difference affects the performance of index pruning and the way to optimize it.

In this chapter we consider several index pruning techniques such as *term pruning* – a complete removal of posting lists of certain terms, *document pruning* – removing certain portions of the posting lists, and the combination of both.

Apart from introducing the *ResIn* architecture, we make the following contributions:

- We show that results caching has important advantages compared to index pruning. In particular, results caching guarantees high cache hit rates with a constant cache capacity

independently of the document collection size;

- We compare the properties of the original query stream and the query stream after a results cache, and show how the differences affect the applicability of index pruning;
- We compare the efficiency of various static index pruning techniques when a pruned index is used separately or in combination with a results cache;
- We propose a different method of combining term and document pruning that outperforms the one presented by [Ntoulas and Cho 2007].

The remainder of this chapter is organized as follows. We first introduce the *ResIn* architecture in Section 6.2. We describe the experimental setup in Section 6.3. We then present our findings for results caching in Section 6.4 including the comparison of the query-logs before and after the results cache in Section 6.4.2. We discuss index pruning and its combination with results caching in Section 6.5 and conclude with Section 6.6.

6.2. *ResIn* Architecture

In a Web search engine, users submit queries through a front-end server. Upon receiving a new query, such a server forwards it to back-end servers for processing. Each of the back-end servers maintains an index for a subset of the document collection and resolves the query against this subset. The index comprises posting lists for all terms in the sub-collection, where each posting list contains *(document reference, term frequency)* pairs. Once the servers finish processing the query, they return results to the front-end server that displays them to the user. A broker machine is usually responsible for aggregating the results from a number of back-end servers, and returning these results to the front-end server.

It is a natural design choice to place at least one other server in between the front-end and the broker to cache final top-k query results as the broker has to send them to the front-end server in any case.

Definition 6.1. Results cache is a fixed-capacity temporary storage of previously computed top-k query results. It returns the stored top-k results if a query is a *hit* or reports a *miss* otherwise.

An important advantage of implementing the results cache is decreasing the number of queries that hit the back-end servers, and thus reducing the number of servers needed to handle the query traffic.

However, the hit rate of the results cache cannot increase beyond the limit imposed by singletons, which often constitute a large fraction of the query traffic. Hence, we look into techniques that enable increasing the hit rate further.

Definition 6.2. Pruned index is a smaller version of the main index, which is stored on a separate set of servers. Such a static[3] pruned index resolves a query and returns the response that is *equivalent* to what the full index would produce or reports a *miss* otherwise.

We call *term pruning* a complete removal of posting lists of certain terms (*e.g.*, stop words removal or the approach described by [Blanco and Barreiro 2007]), whereas *document pruning* refers to ignoring only certain portions of the posting lists (*e.g.*, [Carmel *et al.* 2001]). We consider pruned index organizations with fewer posting lists (term pruning), shorter posting lists (document pruning) or combine both techniques. Thus, the pruned index is typically much smaller than the main index and requires fewer servers to maintain it.

Figure 6.1 shows the *ResIn* architecture where the results cache and the pruned index are placed between the front-end and the broker. In such an architecture, a query is forwarded to the main index only if both the results cache and the pruned index could not answer it, thus substantially reducing the load on the back-end serves.

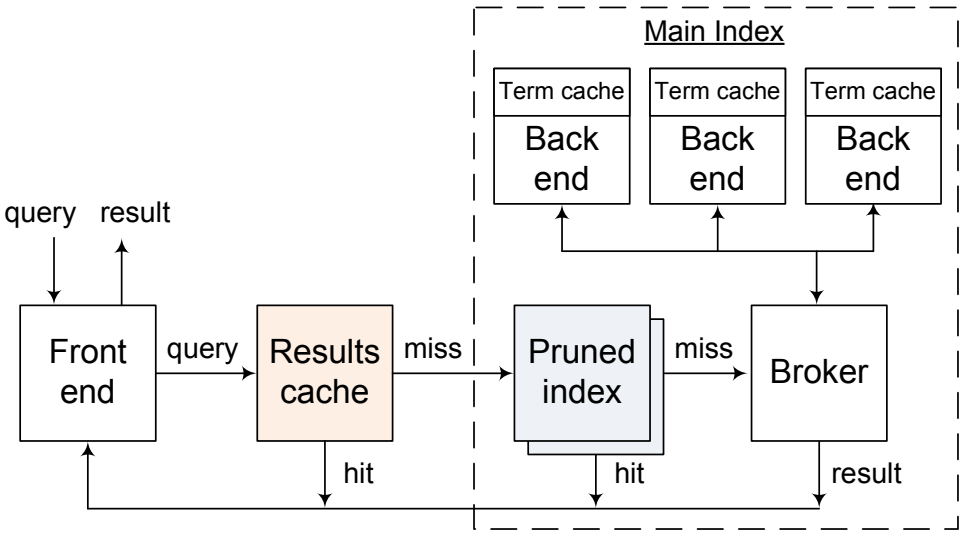

Figure 6.1. Query processing scheme with the *ResIn* architecture.

Having a pruned index in a different network rather than in the one connecting the main index servers is not a good design choice because ensuring the pruned index is up-to-date requires transferring possibly large portions of the index. Thus, we place the pruned index along with the back-end servers holding the main index.

A recent approach presented by [Ntoulas and Cho 2007] employs a similar architecture to reduce the query-rate at the back-end servers without sacrificing the result quality. The authors,

[3] We call it *static* pruning because the pruned version of the index has to be generated in advance, contrary to *dynamic* pruning, which proceeds on a per-query basis saving resources and reducing latency by dynamically skipping non-relevant parts of the index.

however, do not include a results cache, which is a crucial element in our architecture, and employ only a pruned index for this purpose. Note that in reality none of these architectures are really novel as architectures based on clusters for Web search have been proposed before [Brewer 2001]. The importance of the approach proposed by Ntoulas and Cho as well as ours comes from the evaluation of techniques such as caching and pruning.

A typical experimental setup that is used in the literature to investigate the efficiency of an index pruning approach usually considers an original query log or a small set of TREC queries (*e.g.*, [Anh and Moffat 2006; Blanco and Barreiro 2007; Büttcher and Clarke 2006; Carmel *et al.* 2001; Long and Suel 2003; Ntoulas and Cho 2007; Tsegay *et al.* 2007]). In Section 6.4.2 we will show that some statistical properties of query logs change significantly when results caching is used.

6.3. Experimental Setup

Test queries. We used a large query log of the *Yahoo!* search engine. The log contains more than 185M queries submitted at the **.uk** front-end of the search engine. We use this query-log to simulate query processing with a results cache and generate a "miss-log" of queries that are not filtered out by the results cache. Henceforth, we use *all queries* to denote the queries from the original query log, and *misses* to denote the queries from the miss log. Throughout the rest of the chapter we will compare properties of both logs and use them to test various pruning techniques.

Document collection. To investigate the efficiency of index pruning techniques we also used the UK document collection [Boldi *et al.* 2004; Castillo *et al.* 2006]. It contains 77.9M Web-pages crawled from the **.uk** domain in May 2006. We used the Terrier platform[4] to index the collection on 8 commodity machines, which resulted in a distributed compressed index of approximately 40GB without positional information.

6.4. Results Caching

Results cache is a temporary storage for previously computed partial query results. We assume here dynamic caching: when the system processes the query response containing top-k results and returns it to the user, it stores these results in the cache, such that, for future requests of the same query it returns these results instead of asking the back-end servers to process the same query again. Results cache uses an eviction policy to ensure that it never exceeds a maximum capacity. A query produces a *hit* if it was found in the cache and a *miss* otherwise.

In this section we analyze the performance of results caching using a real query log and study the differences between the original query stream and the stream of misses after the results cache.

[4] http://ir.dcs.gla.ac.uk/terrier

6.4.1. Results Cache Performance

To evaluate the performance of the results cache we implemented the LRU policy and tested it with our query log. We varied the cache capacity from $1M$ to $15M$ entries, where each entry contains a query and its top-k results.

Figure 6.2 shows cache-hit rates obtained with different cache capacities for our query log. For each point we warm up the cache with the first $40M$ queries and then measure the average cache hit for the remaining $145M$ queries.

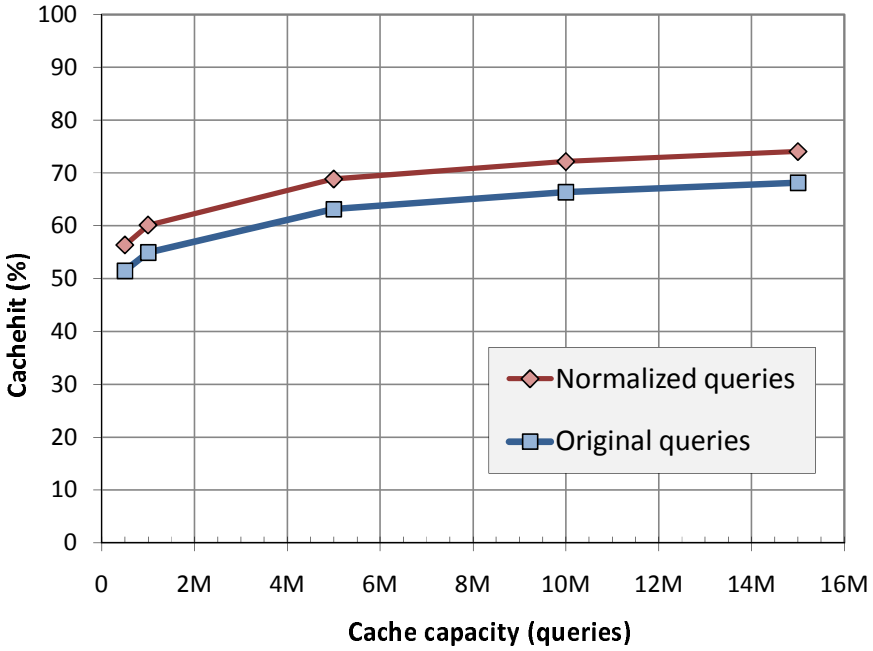

Figure 6.2. Cache hit achieved with a large results cache using the LRU eviction policy.

We also normalized each query by converting it to lower case, removing special symbols and sorting the terms in each query in alphabetical order. Despite the simplicity of this normalization[5], it performs similarly to normalization procedures in current search engines. Figure 6.2 shows that the normalization increases the cache hit by roughly $4-5\%$. This happens because semantically similar, but lexicographically different queries have the same normalized versions, *e.g.*, queries written in a different case, containing symbols ignored by search engines, *etc.*

Due to the presence of singleton queries the performance of any results cache is bounded by the fraction of potential hits $hits_{max} = (1 - \frac{distinct\_queries}{all\_queries})$. For the normalized version of our query log this value is equal to 75.04%.

According to our experiments, sophisticated caching policies such as LFU, SDC [Fagni *et al.* 2006] or AC [Baeza-Yates *et al.* 2007c] substantially outperform LRU when the cache capacity

[5] *E.g.*, it ignores phrase queries and treats URLs as sets of terms.

is small, due to the optimized space usage. However, the improvement is marginal when the cache capacity is big enough such that the hit rate approaches its upper bound. In such a case, LRU becomes the best option due to its simplicity.

Let us assume that one entry in the results cache requires around 2KB in order to store top-20 results including document URLs, titles and snippets. This number can be further reduced by using compression. In this case one machine with 16GB of memory can host a results cache with the capacity of approximately $10M$ queries. Thus, we simulated a $10M$ results cache with the LRU eviction policy and used it to process all $185M$ normalized queries. We warm up the cache with the first $40M$ queries, and then record all query misses for the remaining $145M$ queries. As a result, we obtained a "miss log" containing $41M$ (mostly singleton) queries. Notice that the miss log contains queries that have to be processed by the back-end servers, and that query processing has to be optimized for such queries.

A very important property of the results cache is that its hit rate remains constant as the size of the document collection grows because it depends only on the query log properties, in particular on the number of unique queries. This property makes results caching a very efficient technique to reduce the query load on the back-end servers, as it requires a relatively small amount of storage and processing. The size of a pruned index, however, typically grows with the size of the collection as we discuss further in Section 6.5.

6.4.2. *All Queries* vs. *Misses*

First, we confirm that the average number of terms in a query increases from 2.4 for all queries to 3.23 for misses. Fractions of queries with different number of terms are shown in Figure 6.3. In particular, it shows that the majority of single term queries become hits in the results cache and thus very few of them have to be processed by the index.

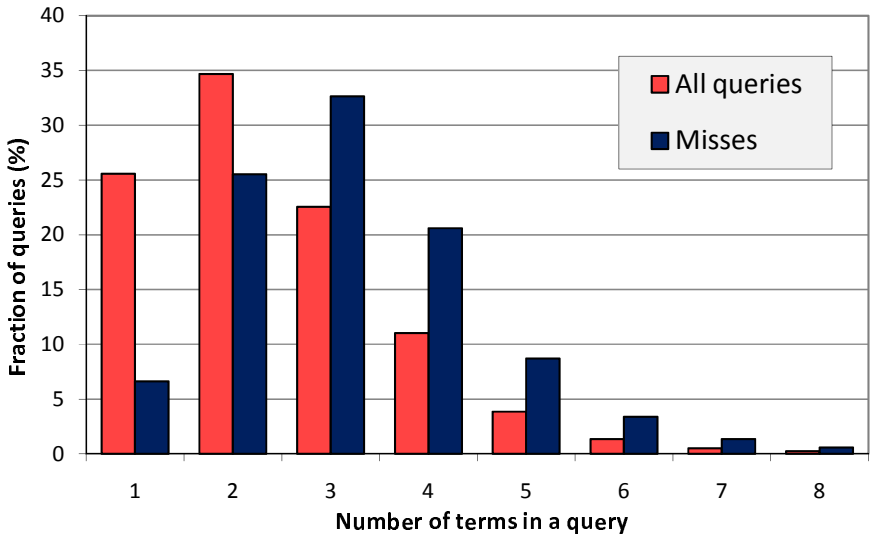

Figure 6.3. Fraction of queries with a given number of terms among *all queries* and *misses*.

Another crucial difference between all queries and misses is the distribution of the sizes of query results. We use the *Yahoo!* search engine to collect the (estimated) results set sizes for 2,000 randomly chosen queries in each set. Figure 6.4 shows that the result set sizes for query misses are approximately two orders of magnitude smaller than for all queries. In particular, almost half of the misses return less than 5,000 results, which is extremely small compared to the total number of documents on the Web indexed by the *Yahoo!* search engine. Figure 6.4 also shows similar results for the (much smaller) UK document collection of 78M documents described in Section 6.3.

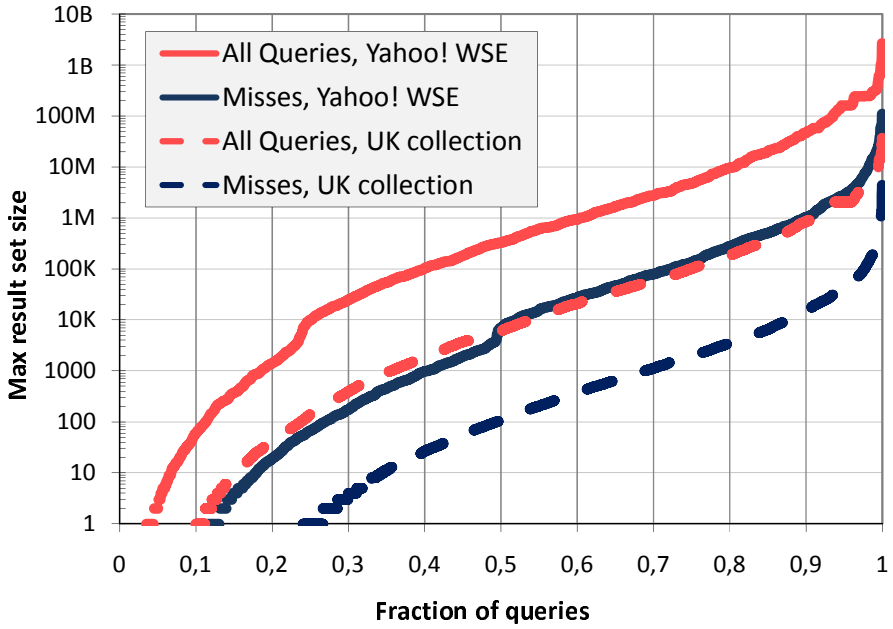

Figure 6.4. Query result size distributions for: 1) the *Yahoo!* Web search engine, and 2) the UK document collection (78M documents).

We say that a query is *discriminative* if it returns relatively few results. Usually it happens because the query contains at least one rare term or the number of terms in the query is large. Both cases suggest that such a query is unlikely to be popular in the query log and therefore results caching performs poorly for discriminative queries. Furthermore, query misses contain a considerable fraction of misspells as they are typically not filtered out by the results cache. In our test set of misses, we detected about 20% of misspelled queries.

We characterize each query term with two properties: popularity and frequency. A term is considered *popular* if it is likely to appear often in *queries*. That is, term popularity is proportional to the number of occurrences of the term in the query log. A term is considered *frequent* if it is likely to appear in *documents*. Frequency (or document frequency) is hence proportional to the number of occurrences of the term in the document collection.

Figure 6.5 shows that terms that are popular in the original query log remain popular in

the miss log as well. We extracted all terms from the original query log and sorted them by popularity. For each term we also computed its popularity in the miss log (0 if the term does not appear there). Each point on the plot shows the average popularity of 1,000 consecutive terms normalized by the size of the query log (185M for the original query log and 41M for the miss log). Notice that averaging over 1,000 consecutive terms generates a smoother curve for misses while having one point per term would compromise visualization.

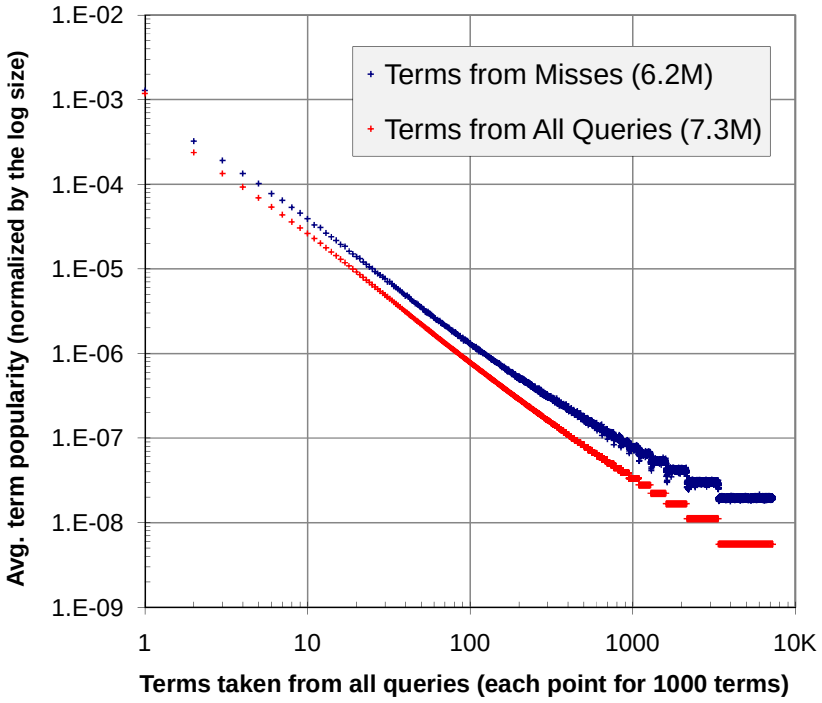

Figure 6.5. Term popularity distribution for *all queries* and for *misses*.

From the figure we conclude that on average popular terms remain popular in the miss log as well. Unpopular terms typically come from queries that are not hits in the results cache, so their absolute popularity is similar in both logs, but the normalized popularity is higher for the miss log because the miss log size is smaller.

This result indicates that, despite significant changes to the query stream after the results cache, techniques that are based on term popularity such as term caching and term pruning work similarly for all queries and for misses. We investigate this observation in detail in the following sections.

6.5. Index Pruning

In this section we consider term and document pruning techniques as well as the combination of both in the context of the architecture of Figure 6.1.

To perform index pruning experiments we used the original query log containing $185M$ normalized queries and the miss log containing $41M$ normalized queries. Since it is computationally expensive to run experiments with such workloads, we randomly selected a test set of $10,000$ queries from the last $5M$ queries in the original query log. Similarly, we used the last $1M$ queries in the miss log to generate a test set of $10,000$ misses. We will refer to these test sets as *all queries* and *misses* in the following sections. The remaining portions of the logs (first $180M$ queries and $40M$ misses) were used to compute term popularities.

6.5.1. Term Pruning

Term pruning selects the most profitable terms and includes their full posting lists into the pruned index. Thus, in order to verify whether a query can be processed by such a pruned index, we only need to check that *all* terms from the query are included in the pruned index. Notice that such a pruning technique assumes that each document in the result set of a query has to contain all terms from the query (conjunctive query processing).

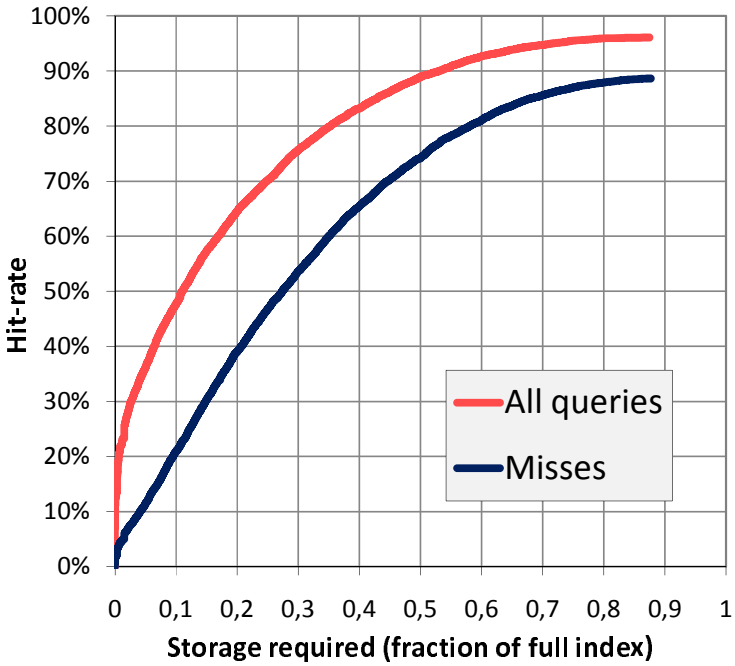

Figure 6.6. Hit rate with the term pruned index.

We implemented a term pruning algorithm that estimates a term profit as: $profit(t) = \frac{pop(t)}{df(t)}$, where $pop(t)$ is the popularity of the term t and $df(t)$ is its document frequency[6]. Hence, the profit of a term is proportional to its popularity and inversely proportional to the space required to store its posting list. We extracted the list of all terms found in the original query log and computed their profits.

[6] In fact, the denominator $df(t)$ in the profit formula implies a non-zero storage cost for t's posting list.

The optimization problem of selecting the terms that maximize the hit rate for a given pruned index size constraint is known to be NP-complete, as it can be reduced to the well known knapsack problem. Thus, we follow the standard heuristics already employed in Chapter 4 that have been shown to perform well in this scenario [Baeza-Yates *et al.* 2007b; Ntoulas and Cho 2007]. We sort the list of terms by profits in descending order and pick the top terms as long as the sum of the posting list sizes for selected terms remains below the given size constraint. Figure 6.6 shows the performance of such a pruning strategy for all queries and for misses.

The hit rate for all queries grows quickly in the beginning because of single term queries and queries that contain few popular terms (typical hits in the results cache). In case of misses though, the growth is nearly linear in the beginning, still being able to handle nearly half of the queries with only a quarter of the index. Thus, even with misses, we can reduce hardware costs by considering a pruned version of the index. For example, suppose that one full cluster processes half of the query load, for some design of index cluster, thus requiring two clusters to process all queries. By using a pruned index, we can have instead one full cluster and one other cluster for the pruned index that is one-fourth of the original cluster.

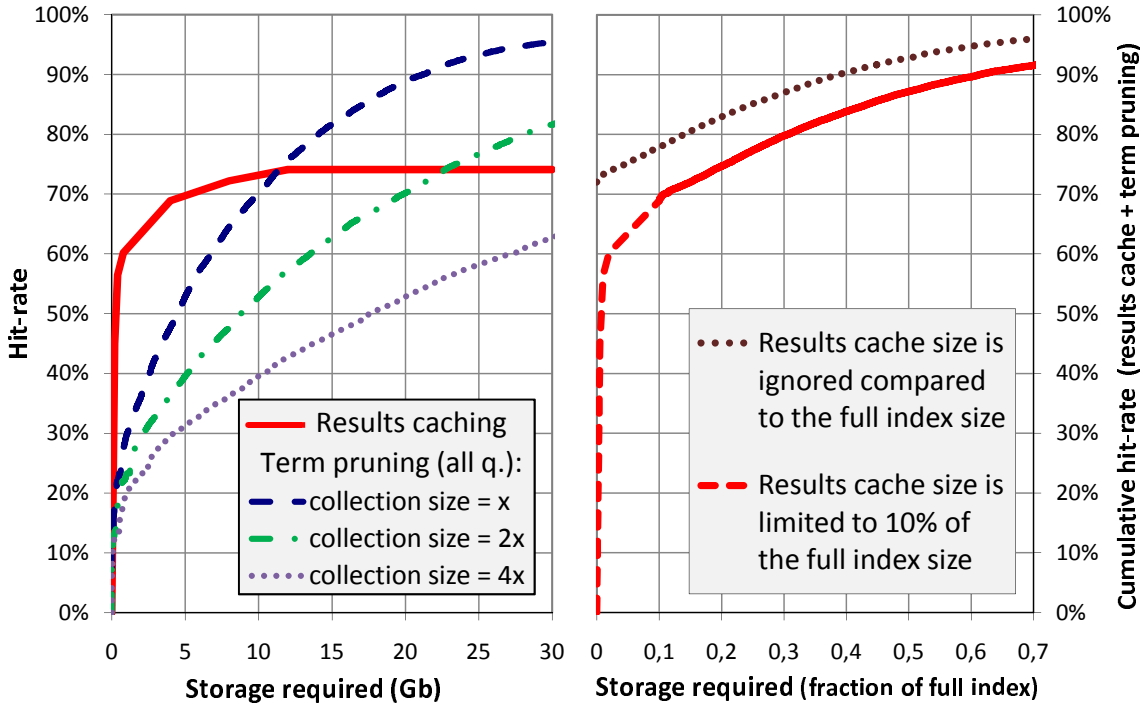

Figure 6.7. Comparison of results caching and term pruning used separately (a), cumulative hit rate with both techniques used together (b).

Figure 6.7-a compares the efficiency of the results cache and the term pruned index when they are used separately. It shows that with a fixed amount of storage available, the hit rate of the pruned index decreases linearly with the growth of the document collection size (a collection of size x corresponds to the set of 78M documents from the .uk domain described in

Section 6.3). The storage required for the results cache, however, depends only on the query log properties and thus remains constant. Therefore, for a Web-scale document collection, the relative results cache size would amount to a small fraction of the full index. Results caching hence is the preferable solution when the amount of available storage is limited and the document collection is large.

However, the hit rate of the results cache is bounded and to avoid this limitation we study the combination of results caching and index pruning. Figure 6.7-b shows the cumulative hit rate obtained by the results cache and the pruned index together following the architecture of Figure 6.1. The top dotted line in Figure 6.7-b shows the maximum cumulative hit rate in such a scenario assuming a sufficiently large index size such that the results cache storage cost can be neglected. We also measured the cumulative hit rate with the real (and relatively small) UK document collection. In this case the results cache size was limited to maximum 10% of the full index size (approx. 4GB). The dashed portion of the bottom line in Figure 6.7-b until 10% of the index size corresponds to the hit rate produced by the results cache and the rest is obtained by the pruned index additionally to the results cache.

Figure 6.7 proves the importance of results caching in our architecture and shows that the combination of results caching and index pruning delivers very good hit rates.

In the next experiment we confirm that a large fraction of misses contains at least one frequent term, although misses typically return few results. Recall that term popularities in the original query log and in the miss log are very similar (see Figure 6.5). Hence, the term pruned index has to include a significant number of popular terms associated with large posting lists.

In Figure 6.8 we use the test set of misses and the UK document collection to plot the correlation between the query result set size and the document frequency of the *most* and the *least* frequent term in a query. The former one is denoted as $MaxDF$ (top plot), while the latter one is $MinDF$ (bottom plot). On both plots, we sort queries by the size of their result set in increasing order, and the dashed line shows the size in log scale. The remaining lines reflect the probability of $MaxDF$ ($MinDF$) being above one of the thresholds: 0, $1K$, $100K$ and $1M$ elements. Each point is computed for 250 consecutive queries with about the same result set size, which can be estimated from the dashed line.

Figure 6.8 suggests that $MinDF$ correlates with the result set size of the query (bottom plot), whereas $MaxDF$ is constantly high for most of the misses (top plot). Hence, the term pruned index has to include large posting lists in order to guarantee a reasonable hit rate.

6.5.2. Document Pruning

Including full (and often large) posting lists in the pruned index might seem redundant, so we studied the document pruning option. Document pruning removes the least important entries from the posting lists. It is based on the observation that if posting lists are sorted such that more relevant documents are stored first, the top-k results for a query are likely to be computed

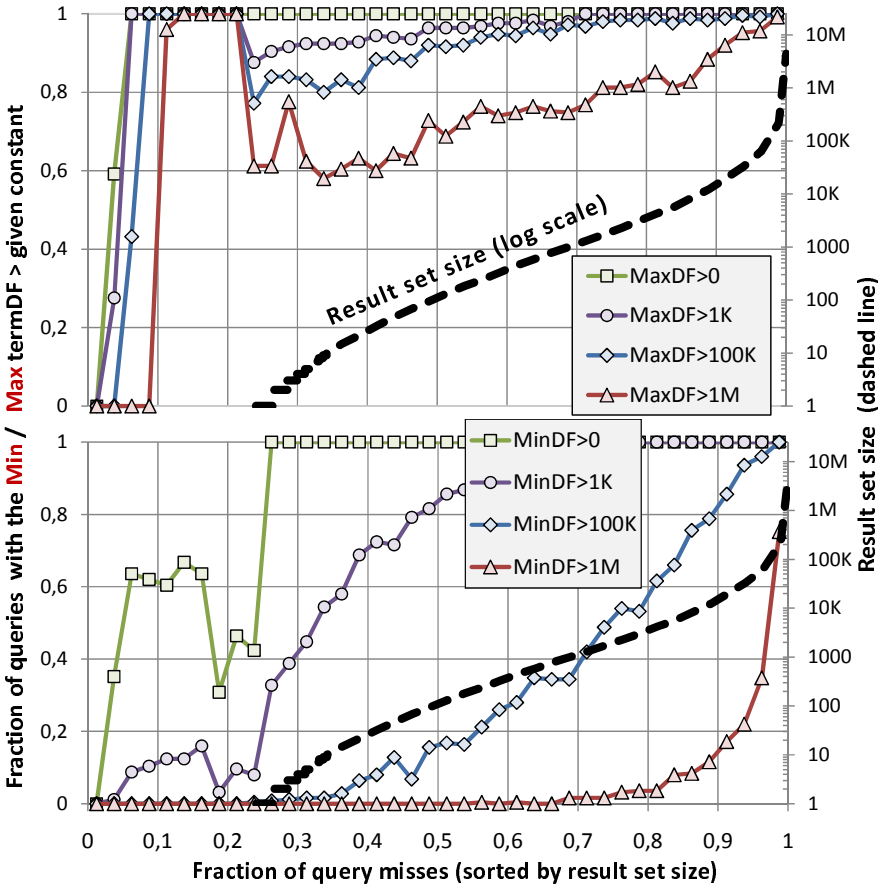

Figure 6.8. Fraction of *misses* with *df* of the most frequent (top) and the least frequent (bottom) term in a query above a given threshold.

without traversing the whole lists, but by examining the top portions of them only [Fagin *et al.* 2001; Ntoulas and Cho 2007]. Thus, posting lists are usually sorted by an attribute that reflects the probability of the document to be included in the top results, such as score, term frequency or impact [Anh and Moffat 2006].

When static document pruning is used, only top-portions of posting lists are included in the pruned version of the index and are used for query processing. A query can be answered from such a pruned index only if it produces exactly the same top-k results as the full index would. It is in general difficult to test this condition, and the only acceptable solution we are aware of is starting processing the query using the pruned index – if the top-k correct results are identified, then the query is a *hit*. Otherwise, the query is a *miss* and it has to be forwarded to and processed again by the main index, thereby affecting latency. To determine the correctness of the results, we can compute a maximum score threshold for all potentially relevant documents whose scores cannot be computed exactly due to truncation. Then, we verify whether all top-k results have scores above the threshold [Fagin *et al.* 2001; Ntoulas and Cho 2007].

Document pruning depends on the ranking function as it significantly influences how quickly the top-k results for a query can be found while examining the top portions of the posting lists. Following [Craswell *et al.* 2005], we used the following weighting formula:

$$score(d,q) = \sum_{\forall t \in q} (bm25(t,d) + \omega \frac{pr(d)}{pr(d)+\kappa}),$$

where $bm25(t,d)$ is the non-normalized BM25 score of the document d for a term t and $pr(d)$ is the query independent score of the document d. To model $pr(d)$ we computed PageRank [Brin and Page 1998] values from the graph of the UK collection [Boldi and Vigna 2004]). We do not have relevance judgments for our document collection to estimate the values of κ and ω, so we fix $\kappa = 1$ and vary the value of ω to study the impact of the PageRank weight on the document pruning performance. Intuitively, the higher the weight ω is, the better document pruning performs. In the following experiments we set ω to 0 (no PageRank), 10 and 20.

Notice that the way the query independent score is included in the final score affects the index construction algorithm. Our formula guarantees that the top-k results obtained from the pruned index and having scores above the maximum score threshold are exactly the same as if they were computed from the full index.

Figure 6.9 shows the fraction of queries whose correct top-10 results are found in all pruned posting lists relevant to the query. The maximum Posting List Length (PLL_{max}) specifies the pruning threshold and is set to $300K$ entries. From the figure, we can see that document pruning works better with non-discriminative queries (*i.e.*, queries with a large result set size, which correspond to the points on the right hand side of the plots). Furthermore, the fraction of queries that require no more than top-$300K$ entries in the posting lists grows with the increase of the PageRank weight ω.

The same plot shows the fraction of queries with the *least* popular term having more than 100 occurrences in the query log. We could see that non-discriminative queries contain popular terms only, while unpopular terms appear in queries that return few results. This observation indicates that popular terms tend to be frequent as well.

Figure 6.9 suggests that document pruning performs much better for all queries than for misses. Indeed, Figure 6.10 shows that while working well for all queries, document pruning requires very high values of ω to outperform term pruning with misses. Each point of Figure 6.10 is obtained by computing the hit rate of the document pruned index with a number of different PLL_{max} values and selecting the one that delivers the maximal hit rate. In fact, we compute an *upper bound* for the hit rate – the fraction of queries for which the correct top-10 results can be found in each pruned posting list relevant to the query. The dashed lines correspond to term pruning for comparison.

We observed that the efficiency of document pruning correlates to the size of the query result. When the query result is small, it is likely that we have to scan large parts of the posting lists (or even whole lists) to obtain enough documents. It can be very inefficient because fre-

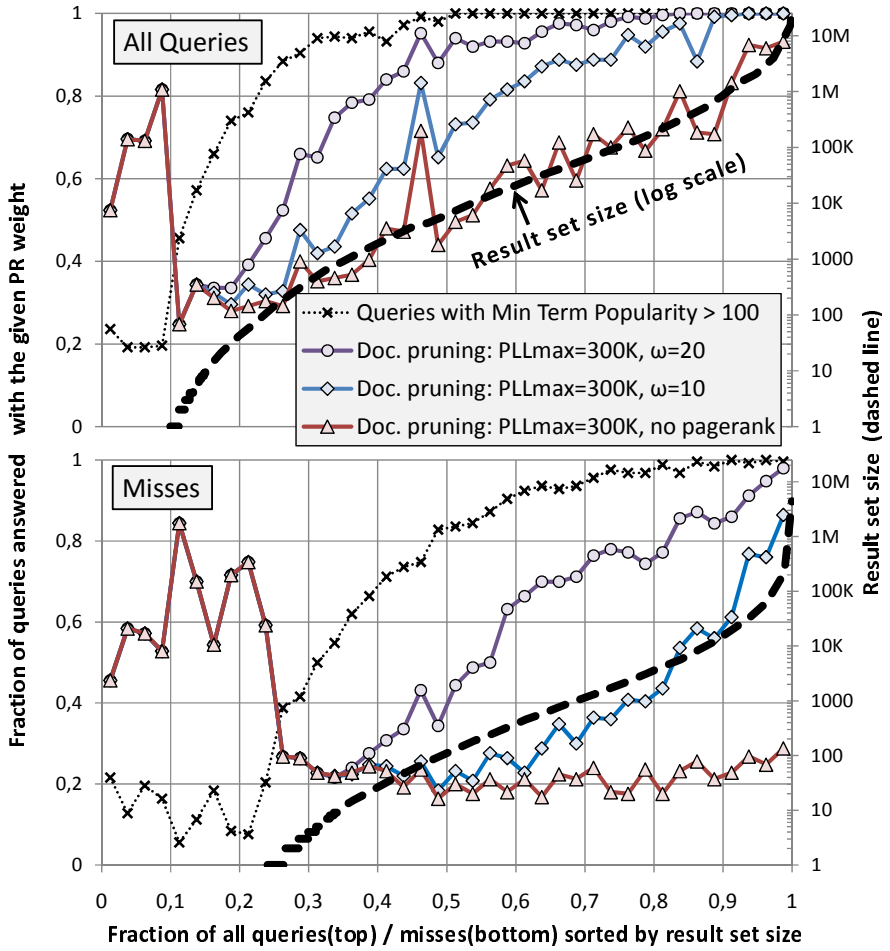

Figure 6.9. Fraction of *all queries* (top) and *misses* (bottom) that can be resolved from posting lists truncated to the $PLL_{max} = 300K$ topmost entries.

quent terms with long posting lists are common in misses (see Figure 6.8). Thus, the worse performance of document pruning with misses can be explained by the fact that the result set sizes are approximately two orders of magnitude smaller compared to all queries (see Figure 6.4).

6.5.3. Term+Document Pruning

Following [Ntoulas and Cho 2007] we study the combination of term and document pruning in our scenario. The intuition behind combining the two is the following: while including profitable terms, only at most top-PLL_{max} entries from each posting list should be copied to the pruned index. If the PLL_{max} constant is chosen such that the hit rate is maximal for a given pruned index size, term+document pruning would deliver at least as good hit rate as term pruning. Indeed, in the case when PLL_{max} is set to ∞ the hit rate with such a pruned index will be exactly the same as for term pruning.

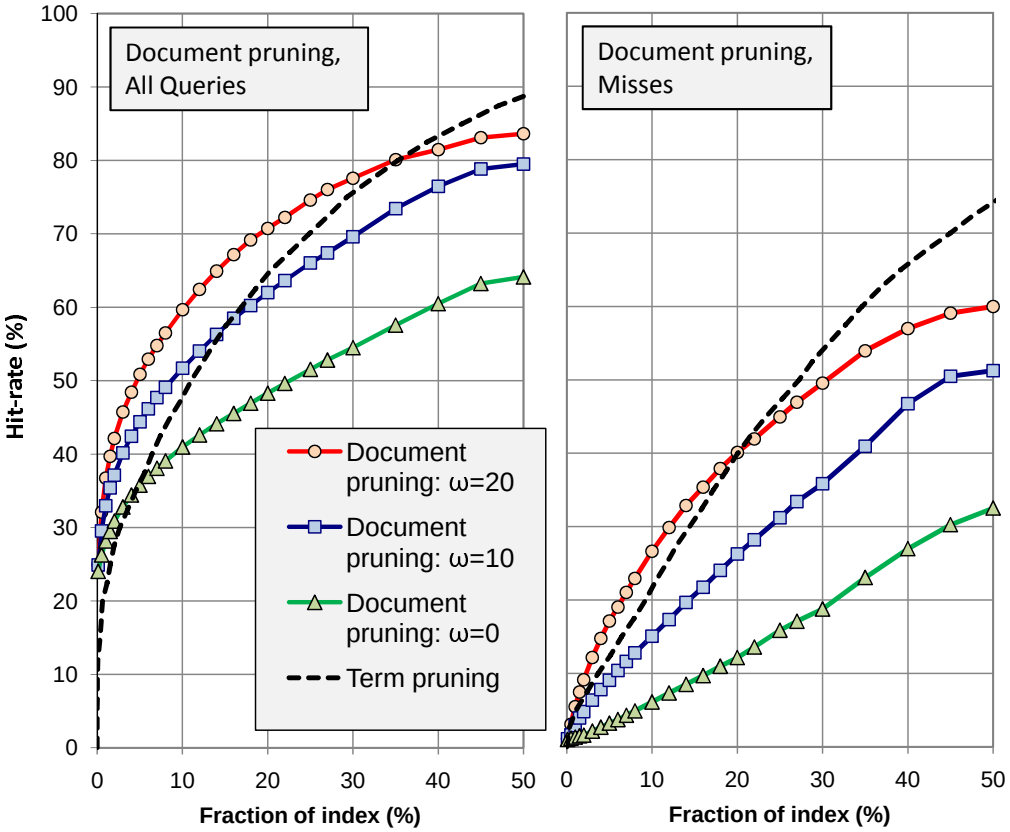

Figure 6.10. Document pruning for *all queries* (left) and *misses* (right) compared with term pruning.

To select terms for such a pruned index we adopt the same profit function as for term pruning $profit_1(t) = \frac{pop(t)}{df(t)}$ because it was used by [Ntoulas and Cho 2007]. However, taking into account the observations made in the previous section, we introduce a new profit function $profit_2(t) = \frac{pop(t)}{min(df(t), PLL_{max})}$ specifically tailored to term+document pruning. The rationale behind it is that the storage cost of including a term t in the pruned index is proportional to $df(t)$ when $df(t) < PLL_{max}$, but depends only on the PLL_{max} parameter if $df(t) \geq PLL_{max}$.

Figure 6.11 shows an upper bound for the hit rate[6] of term+document pruning with the two profit functions defined above for all queries and for misses. Each point is obtained by computing the performance of the term+document pruned index with a number of different PLL_{max} values and selecting the one that delivers the maximal hit rate. Notice that when the maximum hit rate is achieved with $PLL_{max} = \infty$, the corresponding point coincides with the dashed line, which means that there is no improvement over term pruning.

Figure 6.11 suggests that the new $profit_2$ function (left plot) substantially outperforms $profit_1$ (right plot), especially with high PageRank weights. For example, the table below

[6] Same as in Figure 6.10 we compute an *upper bound* of the hit rate by calculating the fraction of queries for which the correct top-10 results can be found in each pruned posting list relevant to the query.

shows the hit rate values obtained with the pruned index 10 times smaller than the full index for both profit functions and different values of ω:

	Term pruning only	Term+document pruning *(upper bounds)*					
		$profit_1$			$profit_2$		
		ω=0	ω=10	ω=20	ω=0	ω=10	ω=20
All queries	47.7	47.7	54.9	61.4	49.7	61.7	**69.3**
Misses	21.7	21.7	21.7	28.6	21.7	26.2	**37.7**

Table 6.1. *ResIn*: Hit rates with 10% pruned index.

Figure 6.11 also shows that for all queries term+document pruning brings substantial benefits compared to term pruning only. However, with misses it yields only a small increase of hit rate when the PageRank weight is very high. For example, the table above shows that the hit rates of term+document pruning with misses noticeably outperform the baseline 21.7% of term pruning with $\omega = 20$ only. Furthermore, term+document pruning induces high processing costs and latencies compared to term pruning, therefore becoming rather unusable with misses.

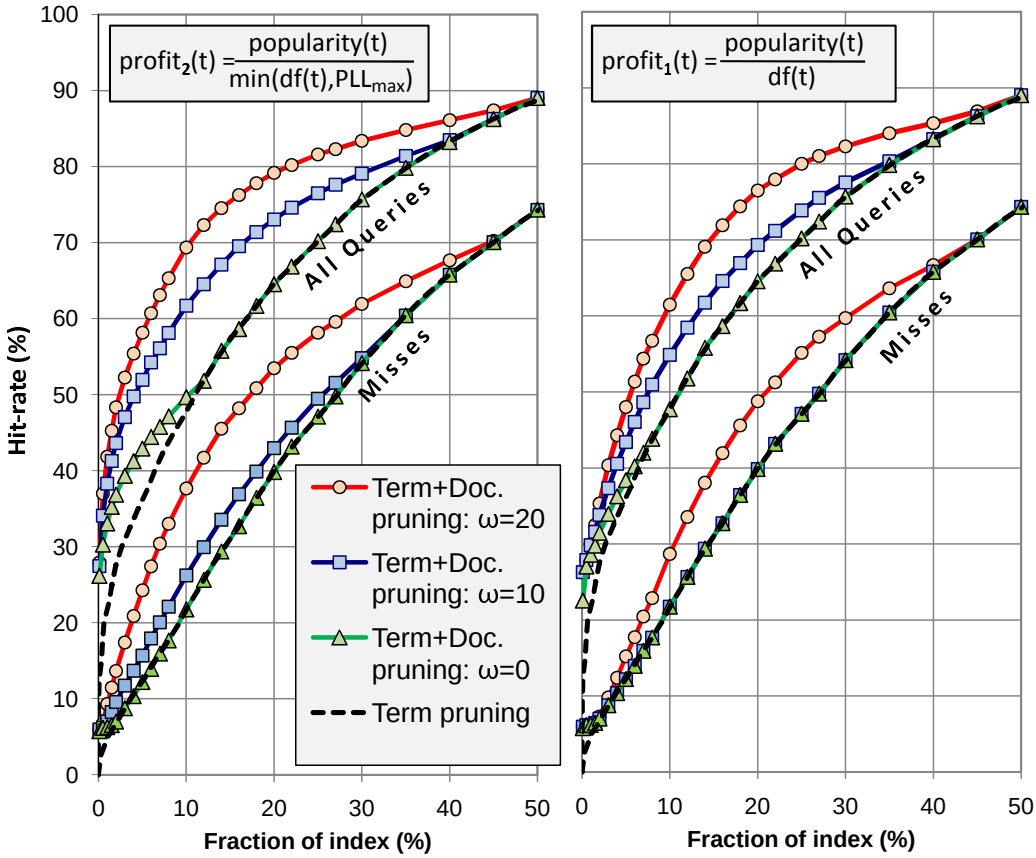

Figure 6.11. Term+document pruning for *all queries* and for *misses* with both profit functions.

To perform the last experiment, we fixed the size of the pruned index to 10% of the full index (*i.e.*, the size of the pruned index grows *linearly* with the size of the document collection) and the PageRank weight ω to 20 (the maximum value in our experiments). We measure the hit rate of the pruned index for fractions of the document collection indexed on 1 and 4 machines respectively, as well as for the full index on 8 machines.

Figure 6.12 shows that the hit rate is approximately the same for the term pruned index when its size is 10% of the full index size (dashed lines) because the length of the posting lists increases linearly with the collection size.

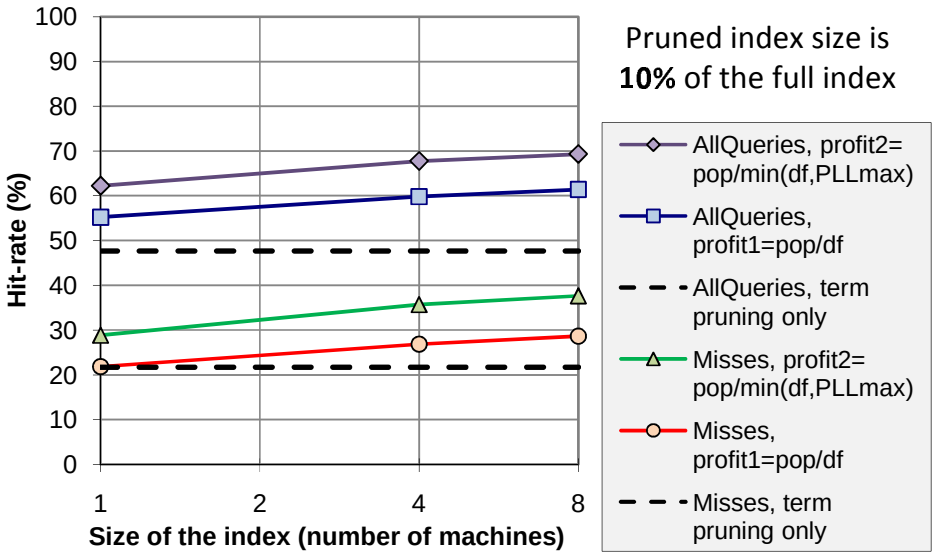

Figure 6.12. Index pruning efficiency for different sizes of the document collection.

The hit rate for term+document pruning grows with the size of the document collection. This happens because document pruning works better with non-discriminative queries, which match a large number of documents. However, in practice, if such a pruned index had to be distributed among several servers following the standard document partitioning scheme, each server would have to compute the top-k results for each query from its (small) fraction of the index. Thus, the resulting hit rate suffers because each server matches a small set of documents in its local index.

6.5.4. Discussion

The original stream of queries contains a large fraction of non-discriminative queries that usually consist of few frequent terms (*e.g.*, navigational queries). Since frequent terms tend to be popular, those queries are likely to repeat in the query log and therefore are typical hits in the results cache. At the same time, these queries would be good candidates to be processed with the document pruned index due to their large result set size. Therefore, document pruning does

not perform well anymore when the results cache is included in the architecture. The same conclusion can be drawn from the fact that misses return much fewer results than original queries on average.

However, individual term popularites are similar for all queries and for misses. Therefore, the contents of the term pruned index does not change much. A larger fraction of short queries explain higher hit rates with the term pruned index for all queries than for misses.

6.6. Conclusions

In this chapter we have presented the *ResIn* architecture for Web search engines that combines results caching and index pruning to reduce the query workload of back-end servers. With *ResIn*, we showed that such a combination is more effective than previously proposed approaches, for example being capable of handling up to 85% of queries with four times less resources than the full index requires.

Results caching is an effective way to reduce the number of queries processed by the back-end servers, because its efficiency is independent on the size of the document collection. Its performance is bounded, however, by the amount of singleton queries in the query stream.

We observed that the results cache significantly changes the characteristics of the query stream: the queries that are misses in the results cache match approximately two orders of magnitude fewer documents on average than the original queries. However, the results cache has little effect on the distribution of query terms.

These observations have important implications for implementations of index pruning: the results cache only slightly affects the performance of term pruning, while document pruning becomes less effective, because it targets the same queries that are already handled by the results cache.

When combining term and document pruning, we substantially increased the hit rates by considering a better term profit function. We have also found that document pruning is more effective when query independent scores, such as PageRank, have high weights in the final scoring formula.

Overall, *ResIn* has allowed us to gain significant insights into the dependence between different techniques for reducing the load on the back-end servers in a Web search engine, and it has allowed us to improve existing techniques by testing them in a more realistic setting.

Chapter 7

Conclusions

7.1. Summary of the Work

Despite the success of Web search, the problem of efficient and effective searching in large-scale data repositories is far from being solved. In particular, improving the quality of search and coping with the rapidly growing amount of information are at the current focus of research. In this thesis we addressed both problems by suggesting novel distributed indexing mechanisms and studying various query processing optimizations.

We investigated highly distributed query processing over document collections (geographically) distributed in a Peer-to-Peer network. We introduced the concept of *query-driven indexing*: an index construction or caching strategy that adapts to the querying patterns expressed by the users. We observed that such strategies are particulary beneficial in a Peer-to-Peer setting where connections between the peers exhibit limited bandwidth and long latencies. Thus, high costs in transmitting the data make caching techniques even more attractive than in centralized architectures such as Web search engines.

We proposed several query-driven indexing strategies that enable efficient query processing in Peer-to-Peer data repositories. The common pattern shared by these approaches is that they are based on a distributed caching system that is maintained in a Peer-to-Peer network without any central coordination.

We proposed an indexing strategy for *efficient XPath query processing* in a structured Peer-to-Peer XML storage. We also designed two approaches for Peer-to-Peer text retrieval. Both of them, the *Distributed Cache Table* and the *QDI approach* enable efficient multi-keyword search over distributed document collections. While the former has certain scalability limitations and is rather suitable for middle-scale digital libraries, the latter is designed for Web-scale Peer-to-Peer information retrieval.

We also explored Web search engine architectures that utilize a centralized caching component. We studied the impact of such a centralized results cache on the efficiency of index-based query processing. In particular, we showed that the performance of index pruning is fundamen-

tally affected by the changes in the query traffic that the results cache induces. We proposed *ResIn* – a variation of the Web search engine architecture that includes both: the results cache and the static pruned index.

To summarize, we observed that geographical distribution of the index affects the applicability of caching. In particular, indexing with term combinations was shown to be beneficial in the P2P setting. On the other hand the presence of a centralized results cache (e.g., in a Web search engine architecture) affects the ways query processing can be optimized. For example, we learnt that queries containing both frequent and unfrequent terms often occur in cache misses and therefore account for a significant fraction of processing performed by the main index.

7.2. Future Work

There are several aspects that could be addressed by future work.

Performance of the Peer-to-Peer indexing techniques under churn. The Peer-to-Peer indexing approaches presented in this thesis do not address the dynamics of the underlying P2P network. While the analysis of the behavior of many P2P systems under churn is available, applications built on top of this systems often assume that peers are relatively stable. We implicitly made the same assumption although it might not hold in many practical situations. An important aspect of the future work would be to test the performance of the presented indexing techniques (especially the QDI approach) under churn.

Using Peer-to-Peer information retrieval to address the long-tail of the query distribution. Search engines are more motivated to optimize the performance of navigational and transactional queries because they better contribute to their advertising model. Such queries are also relatively cheap to process. On the other hand, a P2P-IR system might potentially deliver better results for informational queries than traditional centralized Web search engines due to a different incentive mechanism. An indexing mechanism specifically designed to process "complex" queries potentially with a better quality than Web search engines might be a prerequisite for the long-waited "killer" application of the Peer-to-Peer technology.

Analysis of the impact of index pruning on the efficiency of term caching for Web search engines. In Chapter 6 we learned that while reasoning about Web search engine architectures it is important to consider the interaction between the components. We showed that results caching affects the performance of static index pruning and concluded that they can be combined together. The next step is to asses the impact of index pruning on the performance of term caching at the back-end servers.

Combination of the index partitioning techniques for more efficient query processing for Web search engines. So far document partitioning is preferred by search engines over term partitioning due to better load balancing and lower indexing costs. However, with the growing amount of information and changing hardware specifications hybrid partitioning techniques for Web search engines might be (re-)considered.

Bibliography

K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CoopIS'01: Proceedings of the 9th International conference on Cooperative Information Systems*, Trento, Italy, 2001. @pages 3, 14, 15, 16, 17, 34

K. Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *WDAS'02: Proceedings of the 4th workshop on Distributed Data and Structures*, Paris, France, 2002. @pages 35

K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The Essence of P2P: A Reference Architecture for Overlay Networks. In *P2P'05: Proceedings of the 5th International conference on Peer-to-Peer Computing*, Konstanz, Germany, 2005. @pages 15

S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE'08: Proceedings of the 24th International conference on Data Engineering*, Cancún, México, 2008. @pages 3, 19

K. Albrecht, R. Arnold, M. Gahwiler, and R. Wattenhofer. Join and Leave in Peer-to-Peer Systems: The Steady State Statistics Service Approach. Technical Report 411, ETH Zurich, 2003. @pages 42

V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR'06: Proceedings of the 29th Intrnaltional ACM SIGIR conference on Research and Development in Information Retrieval*, Seattle, WA, USA, 2006. @pages 24, 107, 115

J. Aspnes and G. Shah. Skip Graphs. In *SODA'03: Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, USA, 2003. @pages 17

C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed Query Processing Using Partitioned Inverted Files. In *SPIRE'01: Proceedings of the International Symposium on String Processing and Information Retrieval*, Laguna de San Rafael, Chile, 2001. @pages 20

R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on Distributed Web Retrieval. In *ICDE'07: Proceedings of the 23rd International conference on Data Engineering*, Istanbul, Turkey, 2007a. @pages 2, 4

R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The Impact of Caching on Search Engines. In *SIGIR'07: Proceedings of the 30th International ACM SIGIR conference on Research and Development in Information Retrieval*, Amsterdam, The Netherlands, 2007b. @pages 22, 23, 54, 104, 113

R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission Policies for Caches of Search Engine Results. In *SPIRE'07: Proceedings of the 14th Symposium on String Processing and Information Retrieval*, Santiago, Chile, 2007c. @pages 23, 108

R. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999. @pages 20

W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. DL Meets P2P – Distributed Document Retrieval Based on Classification and Content. In *ECDL'05: Proceedings of the 9th European conference on Research and Advanced Technology for Digital Libraries*, Vienna, Austira, 2005a. @pages 26

W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive Distributed Top-K Retrieval in Peer-to-Peer Networks. In *ICDE'05: Proceedings of the 21st International conference on Data Engineering*, Tokyo, Japan, 2005b. @pages 4

M. Bender, S. Michel, P. Triantafillou, and G. Weikum. Design Alternatives for Large-Scale Web Search: Alexander was Great, Aeneas a Pioneer, and Anakin has the Force. In *LSDS-IR'07: Proceedings of the workshop on Large-Scale Distributed Systems for Information Retrieval*, Amsterdam, The Netherlands, 2007. @pages 21

M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Improving Collection Selection With Overlap Awareness in P2P Search Engines. In *SIGIR'05: Proceedings of the 28th International ACM SIGIR conference on Research and Development in Information Retrieval*, Salvador, Brazil, 2005a. @pages 4, 26

M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. MINERVA: collaborative P2P search. In *VLDB'05: Proceedings of the 31st International conference on Very Large Data Bases*, Trondheim, Norway, 2005b. @pages 3, 26

M. Bergman. The Deep Web: Surfacing Hidden Value. *JEP the Journal of Electronic Publishing*, 7(1), 8 2001. @pages 1

A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM'04: Proceedings of the ACM SIGCOMM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Portland, USA, 2004. @pages 17, 28

B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient Peer-to-Peer Searches Using Result-Caching. In *IPTPS'03: Proceedings of the 2nd International workshop on Peer-to-Peer Systems*, Berkeley, CA, USA, 2003. @pages 28

R. Blanco and A. Barreiro. Static Pruning of Terms in Inverted Files. In *ECIR'07: Proceedings of the 29th European conference on IR Research*, Rome, Italy, 2007. @pages 23, 24, 106, 107

B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), 1970. @pages 18, 27

P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8), 2004. @pages 107

P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW'04: Proceedings of the 13th International World Wide Web conference*, New York, NY, USA, 2004. @pages 116

A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM'04: Proceedings of the 6th ACM International workshop on Web Information and Data Management*, Washington DC, USA, 2004. @pages 18

E. A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4), 2001. @pages 107

S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *WWW'98: Proceedings of the 7th International World Wide Web conference*, Brisbane, Australia, 1998. @pages 116

A. Broder. A taxonomy of Web search. *SIGIR Forum*, 36(2), 2002. @pages 5

S. Büttcher and C. L. A. Clarke. Efficiency vs. Effectiveness in Terabyte-Scale Information Retrieval. In *TREC'05: Proceedings of the Text REtrieval conference*, Gaithersburg, Maryland, USA, 2005. @pages 24

S. Büttcher and C. L. A. Clarke. A Document-Centric Approach to Static Index Pruning in Text Retrieval Systems. In *CIKM'06: Proceedings of the 15th ACM International conference on Information and Knowledge Management*, Arlington, Virginia, USA, 2006. @pages 24, 107

M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured Peer-to-Peer network. In *WWW'04: Proceedings of the 13th International World Wide Web conference*, New York, NY, USA, 2004. @pages 17

M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *Journal of Grid Computing*, 2(1), 2004. @pages 17, 28

J. Callan. Distributed Information Retrieval. In *Advances in Information Retrieval*, pages 127–150. Kluwer Academic Publishers, 2000. @pages 26

P. Cao and Z. Wang. Efficient top-K Query Calculation in Distributed Networks. In *PODS'04: Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, Paris, France, 2004. @pages 27, 84

D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static Index Pruning for Information Retrieval Systems. In *SIGIR'01: Proceedings of the 24th International ACM SIGIR conference on Research and Development in Information Retrieval*, New Orleans, Louisiana, USA, 2001. @pages 23, 24, 106, 107

C. Castillo, D. Donato, L. Becchetti, P. Boldi, S. Leonardi, M. Santini, and S. Vigna. A Reference Collection for Web Spam. *SIGIR Forum*, 40(2), 2006. @pages 107

Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *SIGCOMM'05: Proceedings of the ACM SIGCOMM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Philadelphia, Pennsylvania, USA, 2005. @pages 17

Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In *SIGCOMM'03: Proceedings of the ACM SIGCOMM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Karlsruhe, Germany, 2003. @pages 13

H. Chen, H. Jin, J. Wang, L. Chen, Y. Liu, and L. M. Ni. Efficient multi-keyword search over P2P Web. In *WWW'08: Proceeding of the 17th International World Wide Web conference*, Beijing, China, 2008. @pages 27

C.-W. Chung, J.-K. Min, and K. Shim. APEX: an adaptive path index for XML data. In *SIGMOD'02: Proceedings of the ACM SIGMOD International conference on Management of Data*, Madison, Wisconsin, USA, 2002. @pages 18

B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *VLDB'01: Proceedings of the 27th International conference on Very Large Data Bases*, Roma, Italy, 2001. @pages 18

N. Craswell, S. Robertson, H. Zaragoza, and M. Taylor. Relevance Weighting for Query Independent Evidence. In *SIGIR'05: Proceedings of the 28th International ACM SIGIR conference on Research and Development in Information Retrieval*, Salvador, Brazil, 2005. @pages 116

A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS'02: Proceedings of the 28th International conference on Distributed Computing Systems*, Vienna, Austria, 2002a. @pages 16

A. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. Technical report, Computer Science Department, Stanford University, 2002b. @pages 16

P. Cudré-Mauroux and K. Aberer. A Decentralized Architecture for Adaptive Media Dissemination. In *ICME'02: Proceedings of the International conference on Multimedia and Expo*, Lausanne, Switzerland, 2002. @pages 57

F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *HPDC'03: Proceedings of the 12th International Symposium on High Performance Distributed Computing*, Seattle, WA, USA, 2003. @pages 4, 26

A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS'03: Proceedings of the 29th International conference on Distributed Computing Systems*, Providence, RI, USA, 2003. @pages 36

A. Datta, M. Hauswirth, R. Schmidt, R. John, and K. Aberer. Range queries in trie-structured overlays. In *P2P'05: Proceedings of the 5th International conference on Peer-to-Peer Computing*, Konstanz, Germany, 2005. @pages 15, 16, 38, 53

A. Datta, R. Schmidt, and K. Aberer. Query-load balancing in structured overlays. In *CCGRID'07: Proceedings of the 7th International Symposium on Cluster Computing and the Grid*, Rio de Janeiro, Brazil, 2007. @pages 57

E. S. de Moura, C. F. dos Santos, D. R. Fernandes, A. S. da Silva, P. Calado, and M. A. Nascimento. Improving Web Search Efficiency via a Locality Based Static Pruning Method. In *WWW'05: Proceedings of the 14th International World Wide Web conference*, Chiba, Japan, 2005. @pages 24

R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS'01: Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, Santa Barbara, CA, USA, 2001. @pages 23, 27, 78, 84, 115

T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, 24(1), 2006. @pages 23, 108

L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB'03: Proceedings of the 29th International conference on Very Large Data Bases*, Berlin, Germany, 2003. @pages 18

P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: multi-dimensional queries in P2P systems. In *WebDB'04: Proceedings of the 7th International workshop on the Web and Databases*, Paris, France, 2004. @pages 17

L. Garcés-Erice, P. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross. Data Indexing in Peer-to-Peer DHT Networks. In *ICDCS'04, Proceedings of the 24th International conference on Distributed Computing Systems*, Hachioji, Tokyo, Japan, 2004. @pages 18

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979. @pages 54

S. Girdzijauskas, A. Datta, and K. Aberer. On Small-World Graphs in Non-uniformly Distributed Key Spaces. In *NetDB'05: Proceedings of the International workshop on Networking Meets Databases*, Tokio, Japan, 2005. @pages 15

S. Girdzijauskas, A. Datta, and K. Aberer. Oscar: Small-World Overlay for Realistic Key Distributions. In *DBISP2P'06: Proceedings of the 4th International workshop on Databases, Information Systems and Peer-to-Peer Computing*, Trondheim, Norway, 2006. @pages 17

S. Girdzijauskas, A. Datta, and K. Aberer. Oscar: A Data-Oriented Overlay For Heterogeneous Environments. In *ICDE'07: Proceedings of the 23nd International conference on Data Engineering*, Istambul, Turkey, 2007. @pages 17

O. D. Gnawali. A Keyword Set Search System for Peer-to-Peer Networks, 2002. Master's thesis, Massachusetts Institute of Technology. @pages 28

R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97: Proceedings of the 23th International conference on Very Large Data Bases*, Athens, Greece, 1997. @pages 18

M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *IPTPS'02: Proceedings of the 1st International workshop on Peer-to-Peer Systems*, Cambridge, MA, USA, 2002. @pages 3, 17

H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc., 1978. @pages 86, 96

R. Huebsch, B. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The Architecture of PIER: An Internet-Scale Query Processor. In *CIDR'05: Proceedings of the 2nd Biennial conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2005. @pages 17

S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: a Decentralized Peer-to-Peer Web Cache. In *PODC'02: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterray, CA, USA, 2002. @pages 17

H. Jin and H. Chen. SemreX: Efficient Search in a Semantic Overlay for Literature Retrieval. *Future Generation Computer Systems*, 24(6), 2008. @pages 26

Y.-J. Joung, C.-T. Fang, and L.-W. Yang. Keyword Search in DHT-Based Peer-to-Peer Networks. In *ICDCS'05: Proceedings of the 25th International conference on Distributed Computing Systems*, Columbus, Ohio, USA, 2005. @pages 29

J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *STOC'00: Proceedings of the 32nd ACM Symposium on Theory of Computing*, Portland, Oregon, USA, 2000. @pages 13, 15

F. Klemm, J.-Y. L. Boudec, and K. Aberer. Congestion Control for Distributed Hash Tables. In *NCA'06: Proceedings of the 5th International Symposium on Network Computing and Applications*, Cambridge, MA, USA, 2006. @pages 87, 99

F. Klemm, S. Girdzijauskas, J.-Y. Le Boudec, and K. Aberer. On Routing in Distributed Hash Tables. In *P2P'07: Proceedings of the 7th International conference on Peer-to-Peer Computing*, Galway, Ireland, 2007. @pages 99

K. Kobatake, S. Tagashira, and S. Fujita. A New Caching Technique to Support Conjunctive Queries in P2P DHT. *IEICE Transactions*, 91D(4), 2008. @pages 28

G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *EDBT'04: Proceedings of 9th International conference on Extending Database Technology*, Heraklion, Crete, Greece, 2004. @pages 18

G. Koloniari and E. Pitoura. Peer-to-peer Management of XML Data: Issues and Research Challenges. *SIGMOD Record*, 34(2), 2005. @pages 19

A. Kothari, D. Agrawal, A. Gupta, and S. Suri. Range Addressable Network: A P2P Cache Architecture for Data Ranges. In *P2P'03: Proceedings of the 3rd International conference on Peer-to-Peer Computing*, Linköping, Sweden, 2003. @pages 16

H. Kurasawa, H. Wakaki, A. Takasu, and J. Adachi. Data allocation scheme based on term weight for P2P information retrieval. In *WIDM'07: Proceedings of the 9th ACM International workshop on Web information and data management*, Lisbon, Portugal, 2007. @pages 28

J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *IPTPS'03: Proceedings of the 2nd International workshop on Peer-to-Peer Systems*, Berkeley, CA, USA, 2003. @pages 4, 70

Y. Li, H. V. Jagadish, and K.-L. Tan. Sprite: A Learning-Based Text Retrieval System in DHT Networks. In *ICDE'07: Proceedings of the 23nd International conference on Data Engineering*, Istambul, Turkey, 2007. @pages 29

Y. Li, F. Ma, and L. Zhang. pKSS: An Efficient Keyword Search System in DHT Peer-to-Peer Network. In *ICA3PP'05: Proceedings of the 6th International conference on Algorithms and Architectures for Parallel Processing*, Melbourne, Australia, 2005. @pages 28

K. Lillis and E. Pitoura. Cooperative XPath caching. In *SIGMOD'08: Proceedings of the ACM SIGMOD International conference on Management of Data*, Vancouver, Canada, 2008. @pages 19, 66

X. Long and T. Suel. Optimized Query Execution in Large Search Engines with Global Page Ordering. In *VLDB'03: Proceedings of the 29th International conference on Very Large Data Bases*, Berlin, Germany, 2003. @pages 24, 107

X. Long and T. Suel. Three-Level Caching for Efficient Query Processing in Large Web Search Engines. In *WWW'05: Proceedings of the 14th International World Wide Web conference*, Chiba, Japan, 2005. @pages 23

B. T. Loo, R. Huebsch, J. M. Hellerstein, S. Shenker, and I. Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. In *VLDB'04: Proceedings of the 30th International conference on Very Large Data Bases*, Toronto, Canada, 2004. @pages 29

J. Lu. *Full-text Federated Search in Peer-to-Peer Networks.* PhD thesis, Carnegie Mellon University, USA, 2007. @pages 8, 11, 26

J. Lu and J. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *CIKM'03: Proceedings of the 12th International conference on Information and Knowledge Management*, New Orleans, LA, USA, 2003. @pages 26

J. Lu and J. Callan. Federated Search of Text-Based Digital Libraries in Hierarchical Peer-to-Peer Networks. In *ECIR'05: Proceedings of the 27th European conference on IR Research*, Santiago de Compostela, Spain, 2005. @pages 4, 26

J. Lu and J. Callan. User Modeling for Full-Text Federated Search in Peer-to-Peer Networks. In *SIGIR'06: Proceedings of the 29th International ACM SIGIR conference on Research and Development in Information Retrieval*, Seattle, WA, USA, 2006. @pages 26

E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Survey and Tutorial*, 7, 2005. @pages 15

C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining Query Logs to Optimize Index Partitioning in Parallel Web Search Engines. In *Infoscale'07: Proceedings of the 2nd International conference on Scalable Information Systems*, Suzhou, China, 2007. @pages 23

T. Luu. *Scalable Peer-to-Peer Web search using Highly Discriminative Keys.* PhD thesis, EPFL, Lausanne, Switzerland, 2007. @pages 19, 30, 98

T. Luu, F. Klemm, I. Podnar, M. Rajman, and K. Aberer. ALVIS Peers: A Scalable Full-text Peer-to-Peer Retrieval Engine. In *P2PIR'06: Proceedings of the workshop on Information Retrieval in Peer-to-Peer Networks*, Arlington, VA, USA, 2006. @pages 98, 99

T. Luu, G. Skobeltsyn, F. Klemm, M. Puh, I. Podnar Žarko, M. Rajman, and K. Aberer. AlvisP2P: Scalable Peer-to-Peer Text Retrieval in a Structured P2P Network. In *VLDB'08: Proceedings of the 34th International conference on Very Large Data Bases*, Auckland, New Zealand, 2008. @pages 69, 98

A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel Search using Partitioned Inverted Files. In *SPIRE'00: Proceedings of the 7th International Symposium on String Processing Information Retrieval*, A Coruña, Spain, 2000. @pages 20

B. Mandhani and D. Suciu. Query Caching and View Selection for XML Databases. In *VLDB'05: Proceedings of 31th International conference on Very Large Data Bases*, Trondheim, Norway, 2005. @pages 18

G. S. Manku, M. Bawa, P. Raghavan, and V. Inc. Symphony: Distributed Hashing in a Small World. In *USITS'03: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003. @pages 15

P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS'02: Proceedings of the 1st International workshop on Peer-to-Peer Systems*, Cambridge, MA, USA, 2002. @pages 14, 15

S. Michel. *Top-k Aggregation Queries in Large-Scale Distributed Systems*. PhD thesis, Universität des Saarlandes, Max-Planck-Institut für Informatik, Germany, 2007. @pages 27

S. Michel, M. Bender, N. Ntarmos, P. Triantafillou, G. Weikum, and C. Zimmer. Discovering and Exploiting Keyword and Attribute-Value Co-occurrences to Improve P2P Routing Indices. In *CIKM'06: Proceedings of the 15th ACM International conference on Information and Knowledge Management*, Arlington, Virginia, USA, 2006. @pages 27

S. Michel, P. Triantafillou, and G. Weikum. KLEE: a framework for distributed top-k query algorithms. In *VLDB'05: Proceedings of the 31st International conference on Very Large Data Bases*, Trondheim, Norway, 2005a. @pages 27

S. Michel, P. Triantafillou, and G. Weikum. MINERVA∞: A Scalable Efficient Peer-to-Peer Search Engine. In *Middleware'05: Proceedings of the 6th International Middleware conference*, Grenoble, France, 2005b. @pages 27

T. Milo and D. Suciu. Index Structures for Path Expressions. In *ICDT'99: Proceedings of the 7th International conference on Database Theory*, Jerusalem, Israel, 1999. @pages 18

A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A Pipelined Architecture for Distributed Text Query Evaluation. *Information Retrieval*, 10(3), 2007. @pages 23

W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *WWW'02: Proceedings of*

the 11th International World Wide Web conference, Honolulu, Hawaii, USA, 2002. @pages 3, 16, 47

L. Nguyen, W. G. Yee, and O. Frieder. Adaptive Distributed Indexing for Structured Peer-to-Peer Networks. In *CIKM'08: Proceedings of the 17th ACM International conference on Information and Knowledge Management*, Napa, CA, USA, 2008. @pages 29

A. Ntoulas and J. Cho. Pruning Policies for Two-Tiered Inverted Index with Correctness Guarantee. In *SIGIR'07: Proceedings of the 30th International ACM SIGIR conference on Research and Development in Information Retrieval*, Amsterdam, The Netherlands, 2007. @pages 24, 104, 105, 106, 107, 113, 115, 117, 118

O. Papapetrou, W. Siberski, W.-T. Balke, and W. Nejdl. DHTs over Peer Clusters for Distributed Information Retrieval. In *AINA'07: Proceedings of the 21st International conference on Advanced Networking and Applications*, Niagara Falls, Canada, 2007. @pages 26

Y. Petrakis, G. Koloniari, and E. Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *DBISP2P'04: Proceedings of the 2nd International workshop on Databases, Information Systems, and Peer-to-Peer Computing*, Toronto, Canada, 2004. @pages 18

I. Podnar, T. Luu, M. Rajman, F. Klemm, and K. Aberer. A Peer-to-Peer Architecture for Information Retrieval Across Digital Library Collections. In *ECDL'06: Proceedings of the European conference on research and advanced technology for Digital Libraries*, Alicante, Spain, 2006a. @pages 50

I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Beyond term indexing: A P2P framework for Web information retrieval. *Informatica, Special Issue on Specialised Web Search*, 2006b. @pages 30

I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable Peer-to-Peer Web Retrieval with Highly Discriminative Keys. In *ICDE'07: Proceedings of the 23nd International conference on Data Engineering*, Istambul, Turkey, 2007. @pages 11, 30, 32, 70, 84, 85, 94

M. F. Porter. An Algorithm for Suffix Stripping. *Program*, 14(3), 1980. @pages 58, 71, 88

S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable Content-Addressable Network. In *SIGCOMM'01: Proceedings of the ACM SIGCOMM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Diego, CA, USA, 2001. @pages 3, 13, 14, 34

W. J. Reed. The Pareto, Zipf and Other Power Laws. *Economics Letters*, 74(15-19), 2001. @pages 86

P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Middleware'03: Proceedings of the 4th International Middleware conference*, Rio de Janeiro, Brazil, 2003. @pages 27

S. E. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, and M. Lau. Okapi at TREC. In *TREC'92: Proceedings of the Text REtrieval conference*, Gaithersburg, Maryland, USA, 1992. @pages 72

A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware'01: Proceedings of the International conference on Distributed Systems Platforms*, Heidelberg, Germany, 2001. @pages 3, 14

O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A Peer-to-peer Framework for Caching Range Queries. In *ICDE'04: Proceedings of the 20th International conference on Data Engineering*, Boston, USA, 2004. @pages 16

Sandvine Inc. 2008. Analysis of Traffic Demographics in North-American Broadband Networks, 2008. URL `http://www.sandvine.com/general/documents/Traffic_Demographics_NA_Broadband_Networks.pdf`. @pages 3

M. T. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP – Hypercubes, Ontologies, and Efficient Search on Peer-to-Peer Networks. In *AP2PC'02: Proceedings of the 1st International workshop on Agents and Peer-to-Peer Computing*. @pages 34

S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen. Making Peer-to-Peer Keyword Searching Feasible Using Multi-level Partitioning. In *IPTPS'04: Proceedings of the 3rd International workshop on Peer-to-Peer Systems*, La Jolla, CA, USA, 2004. @pages 29

G. Skobeltsyn and K. Aberer. Distributed Cache Table: Efficient Query-Driven Processing of Multi-Term Queries in P2P Networks. In *P2PIR'06: Proceedings of the workshop on Information Retrieval in Peer-to-Peer Networks*, Arlington, VA, USA, 2006. @pages 49

G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *ODBASE'05: Proceedings of the International Conference on Ontologies, Databases and Applications of SEmantics*, Agia Napa, Cyprus, 2005. @pages 33

G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. ResIn: A Combination of Result Caching and Index Pruning for High-performance Web Search Engines. In *SIGIR'08: Proceedings of the 31st International ACM SIGIR conference on Research and Development in Information Retrieval*, Singapore, 2008. @pages 103

G. Skobeltsyn, T. Luu, I. Podnar Žarko, M. Rajman, and K. Aberer. Query-Driven Indexing for Peer-to-Peer Text Retrieval (poster). In *WWW'07: Proceedings of the 16th International World Wide Web conference*, Banff, Canada, 2007a. @pages 69

G. Skobeltsyn, T. Luu, I. Podnar Žarko, M. Rajman, and K. Aberer. Query-Driven Indexing for Scalable Peer-to-Peer Text Retrieval. In *Infoscale'07: Proceedings of the 2nd International conference on Scalable Information Systems*, Suzhou, China, 2007b. @pages 69, 78, 97

G. Skobeltsyn, T. Luu, I. Podnar Žarko, M. Rajman, and K. Aberer. Web Text Retrieval with a P2P Query-Driven Index. In *SIGIR'07: Proceedings of the 30th International ACM SIGIR conference on Research and Development in Information Retrieval*, Amsterdam, The Netherlands, 2007c. @pages 69

G. Skobeltsyn, T. Luu, I. Podnar Žarko, M. Rajman, and K. Aberer. Query-Driven Indexing for Scalable Peer-to-Peer Text Retrieval. *Future Generation Computer Systems*, 25(1), 2009. @pages 69

P. Skyvalidas, E. Pitoura, and V. V. Dimakopoulos. Replication Routing Indexes for XML Documents. In *DBISP2P'07: Proceedings of the 5th International workshop on Databases, Information Systems and Peer-to-Peer Computing*, Vienna, Austria, 2007. @pages 18

I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01: Proceedings of the ACM SIGCOMM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Diego, CA, United States, 2001. @pages 3, 13, 14, 34

T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *SIGIR'07: Proceedings of the 30th International ACM SIGIR conference on Research and Development in Information Retrieval*, Amsterdam, The Netherlands, 2007. @pages 24

T. Suel, C. Mathur, J.-W. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval. In *WebDB'03: Proceedengs of the International workshop on Web and Databases*, San Diego, CA, USA, 2003. @pages 27, 84, 85

C. Tang and S. Dwarkadas. Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval. In *NSDI'04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, 2004. @pages 28

C. Tang, S. Dwarkadas, and Z. Xu. On Scaling Latent Semantic Indexing for Large Peer-to-Peer Systems. In *SIGIR'04: Proceedings of the 27th International ACM SIGIR conference on Research and Development in Information Retrieval*, Sheffield, UK, 2004. @pages 28

J. Teevan, E. Adar, R. Jones, and M. A. S. Potts. Information Re-retrieval: Repeat Queries in Yahoo's Logs. In *SIGIR'07: Proceedings of the 30th International ACM SIGIR conference on Research and Development in Information Retrieval*, Amsterdam, The Netherlands, 2007. @pages 23

C. Tryfonopoulos, S. Idreos, and M. Koubarakis. Publish/Subscribe Functionalities for Future Digital Libraries using Structured Overlay Networks. In *DELOS'05: Proceedings of the 8th International workshop of the DELOS Network of Excellence on Digital Libraries on Future Digital Library Management Systems*, Schloss Dagstuhl, Germany, 2005. @pages 28

Y. Tsegay, A. Turpin, and J. Zobel. Dynamic Index Pruning for Effective Caching. In *CIKM'07: Proceedings of the 16th ACM Inernational conference on Information and Knowledge Management*, Lisbon, Portugal, 2007. @pages 24, 107

J. Winter. Routing of structured queries in large-scale distributed systems. In *LSDS-IR'08: Proceedings of the 6th Workshop on Large-Scale Distributed Systems for Information Retrieval*, Napa Valley, CA, USA, 2008. @pages 19

J. Xu and B. Croft. Cluster-based Language Models for Distributed Retrieval. In *SIGIR'99: Proceedings of the 22nd International ACM SIGIR conference on Research and Development in Information Retrieval*, Berkeley, CA, USA, 1999. @pages 26

M. R. Yong Yang, Rocky Dunlap and B. F. Cooper. Performance of Full Text Search in Structured and Unstructured Peer-to-Peer Systems. In *INFOCOM'06: Proceedings of the 25th conference on Computer Communications*, Barcelona, Spain, 2006. @pages 29

J. Zhang, X. Long, and T. Suel. Performance of Compressed Inverted List Caching in Search Engines. In *WWW'08: Proceedings of the 17th International World Wide Web conference*, Beijing, China, 2008. @pages 23

J. Zhang and T. Suel. Efficient Query Evaluation on Large Textual Collections in a Peer-to-Peer Environment. In *P2P'05: Proceedings of the 5th International conference on Peer-to-Peer Computing*, Konstanz, Germany, 2005. @pages 4, 21, 27, 70, 84

J. Zhang and T. Suel. Optimized Inverted List Assignment in Distributed Search Engine Architectures. In *IPDPS'07: Proceedings of the 21st International Parallel & Distributed Processing Symposium*, Rome, Italy, 2007. @pages 23

L. Zhang, F. Zou, and F. Ma. KRBKSS – A keyword relationship based keyword-set search system. *Journal of Zhejiang University SCIENCE*, 6A(6), 2005. @pages 28

B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, Berkeley, CA, USA, 2001. @pages 15

C. Zimmer, C. Tryfonopoulos, and G. Weikum. Exploiting correlated keywords to improve approximate information filtering. In *SIGIR'08: Proceedings of the 31st International ACM SIGIR conference on Research and Development in Information Retrieval*, Singapore, 2008. @pages 27

Curriculum Vitæ

Personal Data

Name:	**Gleb Skobeltsyn**
Date of birth:	September 7, 1979
Place of birth:	Moscow, Russia
Nationality:	Russian
Languages:	English (fluent), French (intermediate), Russian (native)

Education

PhD in Computer Science **2008**

Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.
Distributed Information Systems Laboratory, School of Computer
and Communication Sciences. Supervisor – Prof. Karl Aberer.
PhD thesis: Query-Driven Indexing in Large-Scale Distributed Systems.

Graduate studies in Computer Science **2003**

Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.
School of Computer and Communication Sciences.
Project: Visual reasoning about Subject and Aspect Oriented
Programming techniques, grade – 6 (excellent).

M.S. in Computer Science **2002**

Saint-Petersburg State Polytechnical University (SPbSPU), Russia.
Master thesis: Databank of parameters of models of radio
electronic components, grade – excellent. GPA: 4.6/5.

B.S. in Computer Science **2000**

Saint-Petersburg State Polytechnical University (SPbSPU), Russia.
Bachelor thesis: Information system for a construction company,
grade – excellent. GPA: 4.6/5.

Work Experience

Jan. 04 – present	**Research Assistant** at EPFL, Lausanne (Switzerland). Distributed Information Systems Laboratory. Teaching assistantship (5 semesters), student projects supervision, software development for the EU projects Alvis, BRICKS and OKKAM (workpackage leader).
Sep.07 – Dec.07	**Intern** at Yahoo! Research Barcelona (Catalunya, Spain). Research topic: combination of results caching and index pruning for Web search engines.
Mar.99 – Sep.02	**System Administrator** at Electrosfera Ltd., Saint-Petersburg (Russia). Network administration (3 branch offices, 5 servers, 50 clients) and software development.
Jan.97 – Sep.02	**Software Engineer** at Navigator Ltd., Saint-Petersburg (Russia). Project leader and software developer in the following projects: - Information system of a small office enterprise; - Regional databank of children adoption; - Software for automatic parameters calculation of analog elements; - PalmOS dashboard for industrial automatic control system of gas drying; - CAD for window and door blocks; - *etc.*

Publications

Conference and Workshop Papers

1. Sheila Kinsella, Adriana Budura, Gleb Skobeltsyn, Sebastian Michel, John G. Breslin, Karl Aberer: *From Web 1.0 to Web 2.0 and back – How did your Grandma use to tag?* Proceedings of the 10th ACM International Workshop on Web Information and Data Management (**WIDM**), October 30, 2008, Napa, USA.

2. Toan Luu, Gleb Skobeltsyn, Fabius Klemm, Maroje Puh, Ivana Podnar Žarko, Martin Rajman, Karl Aberer: *AlvisP2P: Scalable Peer-to-Peer Text Retrieval in a Structured P2P Network (demo paper).* Proceedings of the 34th International Conference on Very Large Data Bases (**VLDB**), August 24 – 30, 2008, Auckland, New Zealand.

3. Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, Ricardo Baeza-Yates: *ResIn: A Combination of Result Caching and Index Pruning for High-performance Web Search Engines.* Proceedings of the 31st International ACM **SIGIR** Conference, July 20 – 24, 2008, Singapore.

4. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Žarko, Martin Rajman, Karl Aberer: *Web Text Retrieval with a P2P Query-Driven Index.* Proceedings of the 30st International ACM **SIGIR** Conference, July 23 – 27, 2007, Amsterdam, The Netherlands.

5. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Žarko, Martin Rajman, Karl Aberer *Query-Driven Indexing for Scalable Peer-to-Peer Text Retrieval.* Proceedings of the 2nd International Conference on Scalable Information Systems (**Infoscale**), June 6 – 8, 2007, Suzhou, China.

6. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Žarko, Martin Rajman, Karl Aberer *Query-Driven Indexing for Peer-to-Peer Text Retrieval (poster).* Proceedings of 16th International World Wide Web Conference (**WWW**), May 8 – 12, 2007, Banff, Canada.

7. Gleb Skobeltsyn, Karl Aberer: *Distributed Cache Table: Efficient Query-Driven Processing of Multi-Term Queries in P2P Networks.* Proceedings of the CIKM Workshop on Information Retrieval in Peer-to-Peer Networks (**P2PIR**), November 11, 2006, Arlington, USA.

8. Gleb Skobeltsyn, Manfred Hauswirth, Karl Aberer: *Efficient Processing of XPath Queries with Structured Overlay Networks.* Proceedings of the International Conference on Ontologies, Databases and Applications of SEmantics (**ODBASE**), October 31 – November 4, 2005, Agia Napa, Cyprus.

9. Pavel Balabko, Gleb Skobeltsyn, Alain Wegmann *Role Composition in Requirements Engineering: the Method and Prototyping Tool.* Proceedings of 16th International Conference on Advanced Information Systems Engineering (**CAiSE**), June 7 – 11, 2004, Riga, Latvia.

Journal Papers

1. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Žarko, Martin Rajman, Karl Aberer: *Query-Driven Indexing for Scalable Peer-to-Peer Text Retrieval.* Future Generation Computer Systems, Volume 25, Issue 1, January 2009.

2. Alexander Isakov, Kirill Skobeltsyn, Gleb Skobeltsyn, Aleksey Polkovnikov: *DISP dialog system for electronic circuits design.* EDA Express journal (in Russian), N5, 2001, Moscow, Russia.

3. Alexander Isakov, Kirill Skobeltsyn, Gleb Skobeltsyn: *Software for automatic parameters calculation of analog elements.* EDA Express journal (in Russian), N4, 2001, Moscow, Russia.

4. Gleb Skobeltsyn: *FontEditor- Editor for system fonts in VGA text mode.* Prologue journal (in Russian), 1994, Computer science - special issue, Saint-Petersburg, Russia.

Main Awards

1995	First place in Saint-Petersburg school programming Olympiad.
2002	Doctoral School scholarship from EPFL.

Other

1. Co-organizer of the 6th International workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'08) collocated with CIKM 2008.
2. PC member: SAC'08, SAC'09.
3. Reviewer: TKDE journal, FGCS journal.
4. External reviewer: ICDE, VLDB, SIGMOD, P2P conference, *etc.*

Lausanne, October 24, 2008

✠